JOHN J. BARTON
LEE R. NACKMAN

Scientific and Engineering C++

An Introduction with Advanced Techniques and Examples

Scientific and Engineering C++

AN INTRODUCTION with ADVANCED TECHNIQUES and EXAMPLES

John J. Barton
Lee R. Nackman

IBM Thomas J. Watson Research Center
Yorktown Heights, New York

ADDISON-WESLEY
An imprint of Addison Wesley Longman, Inc.
Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Editor: Tom Stone/Debbie Lafferty
uction Supervisor: Helen Wythe
Superscript Editorial Production Services
: Windfall Software (Paul C. Anagnostopoulos/Joe Snowden, using ZzTEX)
Wilson Graphics & Design (Kenneth J. Wilson)
n: Marshall Henrichs
ing Manager: Roy Logan

ures and applications presented in this book have been included for their in-
value. They have been tested with care but are not guaranteed for any purpose.
her does not offer any warranties or representations, nor does it accept any lia-
respect to the programs and applications.

e designations used by manufacturers and sellers to distinguish their products
as trademarks. Where those designations appear in this book, and Addison-
s aware of a trademark claim, the designations have been printed in initial caps

Congress Cataloging-in-Publication Data
n J.
fic and engineering C++ : an introduction with advanced
es and examples / John J. Barton, Lee R. Nackman.
cm.
es bibliographical references and index.
)-201-53393-6
(Computer program language) I. Nackman, Lee R.

.C153B38 1994
–dc20 93-40343
CIP

1994 by Addison Wesley Longman, Inc.

served. No part of this publication may be reproduced, stored in a
stem, or transmitted in any form or by any means, electronic,
photocopying, recording, or otherwise, without prior written
of the publisher. Printed in the United States of America. Published
sly in Canada.

10 MA 02 01 00 99
January 1999

for

Ava Nackman

and

Cynthia Butler

PREFACE

LIKE many scientists and engineers, much of our work involves writing computer programs. Recently we have been writing those programs in C++. We think that our programs are better and that we can do better science and engineering with these programs because they are written in C++. We think you should try C++, and we wrote this book to help you get started.

C++ is one of several new languages that use a programming style called object-oriented programming. To write large programs that are correct, readable, modifiable, affordable, and efficient requires the same creative effort and persistence characteristic of other endeavors in science and engineering. Traditional programming languages, including FORTRAN and C, force us to communicate with the computer in a demeaningly simplistic manner. C++ and an object-oriented programming style elevate the communication to a more abstract level: They provide means for investing intellectual effort to produce better-quality programs and thus better-quality science and engineering, from a given programming project.

Learning C++ will be exciting. Although most of the programming ideas used in languages like FORTRAN, PASCAL, and C are still used in object-oriented programs, the new concepts reorganize the work. Like all new fields, object-oriented programming will seem foreign and exotic. C++ embodies a decade of new ideas from computer science backed up by practical experience. These new ideas will stimulate your thinking about programming and its role in your work. We hope you will find, as we have, that this new view changes programming from a tedious, albeit engaging, process to an intellectual enterprise more comparable to the processes we employ in other scientific and engineering work.

Purpose

The purpose of this book is to teach you how to use C++ and the object-oriented programming style to produce better-quality programs, with an emphasis on scientific and engineering programs. Most such programs today are written in FORTRAN or C and without the benefit of any particular programming methodology. For small programs of strictly numerical content, FORTRAN or C may be adequate. However, larger programs and programs containing nonnumerical code are too expensive to understand, to revise, and to improve if written in FORTRAN or C. We present object-oriented programming as a design and

programming style that addresses these problems and C++ as a programming language designed to allow efficient use of the object-oriented style. If you are still using FORTRAN or C in your programming, we invite you to explore a new world, the world of object-oriented programming in C++.

Audience

Our book teaches object-oriented programming in C++, using examples from science and engineering. It is not a book about scientific computing or numerical analysis nor an introduction to programming. The book moves rapidly through the basic features and syntax of C++, material readily assimilated by an engineer or scientist experienced in programming or, indeed, by any experienced programmer. Our aim is to move quickly beyond syntax and rules to the more interesting and important concepts and techniques of object-oriented programming in C++. The latter part of the book applies the concepts and techniques developed to substantive examples. The examples are drawn primarily from science and engineering, but the concepts and techniques are broadly applicable.

We expect the book to be useful to three (overlapping) groups:

- Engineers and scientists who are experienced programmers in FORTRAN or C.
- Professional programmers experienced in C or C++ looking for a new systematic discussion of object-oriented programming in C++.
- C++ programmers interested in advanced examples useful as a basis for scientific and engineering programming.

In addition to programming experience, some of the examples assume the mathematical maturity typical of an undergraduate student in an engineering or scientific field.

Learning C++ and object-oriented programming will be a challenge regardless of your background. We were frankly amazed that computer programming could be so different. We hope you find this challenge stimulating and rewarding on its own; we are confident that once you understand C++ and object-oriented programming, you will not be satisfied with less.

Acknowledgments

This book was made possible by the considerable patience of our employer, the Research Division of the IBM Corporation, and the personal patience and encouragement of our managers, colleagues, friends, and families. We began work

on this book when we were in the Physical Sciences (Barton) and Manufacturing Research (Nackman) departments. Our managers in those departments—Read McFeely, Franz Himpsel, and Bruce Scott; Mike Wesley, Warren Grobman, and Russ Lange—supported and encouraged our work. A special thanks to the late Mike Wesley, manager, mentor, and friend for a decade: He recognized the importance of producing quality software for engineering applications and provided the environment, encouragement, and support for learning something about how to do it. We have completed work on the book in the Computer Science department, where we enjoy the considerable support and encouragement of our manager Mark Wegman.

We are indebted to all of our colleagues at IBM Research for having made it a special place to work and learn. We especially thank Michael Karasick and Derek Lieber for helping us, over many years, to learn C++ and how to use it, and Louis Terminello for timely and gracious encouragement. We also thank Bjarne Stroustrup and the developers of IBM's C++ compiler, especially Mark Mendell, Dave Streeter, and Ernest Choi, for correspondence and encouragement while we learned and relearned C++.

We were also fortunate to have the help of many reviewers; their comments improved many aspects of the book, ranging from typography to the book's organization. The comments of James Coplien, Tom Lyons, and William Press had an especially large impact on the book. As deadlines loomed, Michael Karasick read furiously through several drafts to help us weed out the worst confusions. We also thank John L. Bradberry, Goodwin Chin, Marshall Cline, Chris Codella, Margaret Ellis, Martin Giles, Franklin Gracer, Peter Juhl, Derek Lieber, Mark Linton, Tom Linton, Stanley Lippman, Alistair McClean, John Morar, Dean Pentcheff, V. T. Rajan, John Rehr, Chris Seekamp, Steve Stevenson, Bjarne Stroustrup, Bob Sutor, Dave Tolle, Hank Walker, and Robert Wang for their many suggestions. The efforts of all these people spared you the early drafts of the book.

Debbie Lafferty, our editor at Addison-Wesley, gently prodded and encouraged us at each step of the way, carefully balancing between pushing too little and too hard.

The love of our families—Ava, Rachel, Samuel, and Joel Nackman; Cynthia Butler and John Anthony and Andrew Butler Barton—has been essential. We thank them for their patience and understanding during all those times when working on the book took time away from them.

CONTENTS

Preface v

PART I Getting Started 1

Chapter 1 Introduction 3

1.1 Object-Oriented Programming 4
1.2 Why C++? 5
1.3 What About . . . ? 6
1.4 Program Design 7
1.5 Organization of the Book 8
1.6 Source Code 9
1.7 Contacting the Authors 9
1.8 Notes and Comments 10
1.9 Exercises 11

Chapter 2 Basics for FORTRAN Programmers 13

2.1 A First Program 13
2.2 Variables, Objects, and Types 18
2.3 C++ Fundamental Types and Operations 21
2.4 Input and Output 27
2.5 Operator Precedence and Associativity 30
2.6 if Statements 32
2.7 Loops 34
2.8 Declarations 38
2.9 Arrays 40
2.10 Pointers 43
2.11 Pointers and Arrays 44
2.12 const Pointers and Pointers to const Objects 46
2.13 Runtime Array Size 47
2.14 Character Strings 50
2.15 References 50
2.16 Functions 52

2.17 Notes and Comments 61
2.18 Exercises 63

apter 3 **Basics for C Programmers 67**
3.1 A First Program 67
3.2 Variables, Objects, and Types 68
3.3 C++ Built-In Types and Operations 70
3.4 Operator Precedence and Associativity 71
3.5 Input and Output 71
3.6 Declarations 75
3.7 Pointers 76
3.8 Memory Management 77
3.9 References 78
3.10 Functions 80
3.11 Notes and Comments 83
3.12 Exercises 84

apter 4 **Classes 85**
4.1 Two Simple Classes 85
4.2 An Array Class 95
4.3 Class Templates 100
4.4 Function Templates 104
4.5 Exceptions 105
4.6 Nested Classes 110
4.7 Overview of C++ Programs 113
4.8 Notes and Comments 114
4.9 Exercises 116

apter 5 **Functions 119**
5.1 Declarations and Definitions 119
5.2 Function Declarations 121
5.3 Function Arguments 124
5.4 Function Return Types 131
5.5 Overloaded Functions 132
5.6 Function Templates 136

5.7 Notes and Comments 139
5.8 Exercises 140

Chapter 6 Functions and Classes 143
6.1 Member Functions and Overloading 143
6.2 Initialization 145
6.3 Copying 148
6.4 Conversion 150
6.5 Operator Functions 159
6.6 Assignment 163
6.7 Special Operators 164
6.8 Destruction 165
6.9 Static Member Functions 166
6.10 Friend Functions 167
6.11 Input/Output Operators for Classes 171
6.12 Notes and Comments 172
6.13 Exercises 173

Chapter 7 Object Lifetime and Memory Management 175
7.1 Object Life Cycle 175
7.2 Object Lifetime 176
7.3 Static Objects 178
7.4 Automatic Objects 185
7.5 Dynamic Objects 188
7.6 Preventing Dangling References and Garbage 190
7.7 Notes and Comments 194
7.8 Exercises 196

Chapter 8 An Example Program 199
8.1 The Problem: Representing a Mesh 199
8.2 Solution One: Arrays 202
8.3 Abstraction and Encapsulation 206
8.4 Solution Two: Introducing Classes 208
8.5 Solution Three: Information Hiding 212
8.6 Notes and Comments 218
8.7 Exercises 219

PART II Expressing Commonality 223

Chapter 9 Expressing Common Behavior 225

9.1 Example: Instrument Control 225
9.2 Classes and Objects 227
9.3 Interfaces and Interface Categories 231
9.4 Sketching Objects and Classes 238
9.5 Create Objects; Use Interfaces 240
9.6 Interface Base Classes 242
9.7 Multiple Interfaces 243
9.8 Using Interfaces as Components 246
9.9 Creating and Using Arrays of Interfaces 250
9.10 Exceptions and Interfaces 257
9.11 Summary 259
9.12 Notes and Comments 260
9.13 Exercises 263

Chapter 10 Expressing Common Implementation 265

10.1 Extension Using Public Inheritance 266
10.2 Extension of Interfaced Classes Using Public Inheritance 270
10.3 Problems with Public Inheritance 274
10.4 Member Function Forwarding 277
10.5 Private Inheritance for Implementation 280
10.6 Mechanics of Inheritance 284
10.7 Base Class Composition 294
10.8 The Meaning of Declarations in a Class 297
10.9 Summary 298
10.10 Notes and Comments 299
10.11 Exercises 301

Chapter 11 Expressing Common Structure 303

11.1 Commonality Expressed by Templates 303
11.2 Templates and Inheritance 307
11.3 Example: Array Classes 311
11.4 Interfaced Array Classes 313
11.5 Global Function Templates 325

11.6 Summary 326
11.7 Notes and Comments 327
11.8 Exercises 327

Chapter 12 **Types 329**
12.1 Basic Ideas of Type 330
12.2 Types and Interfaces 334
12.3 Type Conversions 338
12.4 Loss of Type Information 340
12.5 Types and Class Templates 346
12.6 Restricted Template Expansion 351
12.7 Function-Structure Categories 354
12.8 Summary 355
12.9 Notes and Comments 356
12.10 Exercises 358

PART III Applications and Techniques 361

Chapter 13 **Arrays 363**
13.1 Using Concrete Arrays 364
13.2 Concrete Arrays 368
13.3 Concrete Array References 379
13.4 Concrete Array Projections 388
13.5 Interfaced Arrays 396
13.6 Projections of Interfaced Arrays 403
13.7 Iterators 406
13.8 Summary 413
13.9 Notes and Comments 414
13.10 Exercises 415

Chapter 14 **Pointer Classes 419**
14.1 Pointer Uses 419
14.2 Referential Aggregation 421
14.3 Programmer-Defined Pointer Classes 424
14.4 Copied-Object Pointers 426
14.5 Counted-Object Pointers 431
14.6 Using Counted-Object Pointers 437

14.7 Interface Pointer Classes 441
14.8 Summary 445
14.9 Notes and Comments 446
14.10 Exercises 447

Chapter 15 Classes for Code Organization 451
15.1 The Organization of LAPACK 451
15.2 Grouping Data and Functions 455
15.3 Indefinite Types with Runtime Failures 459
15.4 Expressing Common Structure 464
15.5 Summary 468
15.6 Notes and Comments 469
15.7 Exercises 470

Chapter 16 Algebraic Structure Categories 473
16.1 Algebraic Structures 474
16.2 Example: ComplexFloat 478
16.3 Categories for Single-Set, Single-Law Structures 480
16.4 Categories for Two-Set Structures 492
16.5 Example: Dimensional Analysis 493
16.6 Example: Arrays with Arithmetic 500
16.7 Summary 504
16.8 Notes and Comments 505
16.9 Exercises 505

Chapter 17 Function Objects 511
17.1 Function Pointers 511
17.2 Member Function Pointers 513
17.3 Function Objects 514
17.4 Functional Interface Categories 516
17.5 Function Objects for Delayed Evaluation 518
17.6 Example: A Simple Function Evaluator 526
17.7 Functions over Collections 533
17.8 Summary 535
17.9 Notes and Comments 535
17.10 Exercises 536

Chapter 18 Using Legacy Libraries 539
18.1 Working with C 540
18.2 Example: String Class 542
18.3 Working with FORTRAN 553
18.4 Exploiting Name Commonality in Wrapped FORTRAN 55
18.5 Packed Array Representations 559
18.6 Matrices Implemented with BLAS 563
18.7 Singular Value Decomposition with LAPACK 569
18.8 Summary 577
18.9 Notes and Comments 577
18.10 Exercises 578

Chapter 19 Data Modeling in C++ 583
19.1 Introduction 583
19.2 Classes for Experimental Data 584
19.3 Classes for Linearized Nonlinear Equations 587
19.4 Classes for Automatic Derivatives 589
19.5 Example: Iterative Solution of Nonlinear Equations 594
19.6 Classes for Damped SVD Equations 603
19.7 The main() Program 610
19.8 Summary 612
19.9 Notes and Comments 613
19.10 Exercises 614

References 617

Index 625

Source File Index 667

PART I

Getting Started

CHAPTER 1

Introduction

COMPUTERS perform complex tasks by executing many simple instructions. Humans specify these instructions by writing programs in a programming language. Programming languages combine individual machine instructions into more powerful compound instructions. They also allow new compound instructions to be created, named, and used as units. Computations expressed in these units communicate the program in a more abstract way, ignoring the detail of machine instructions and highlighting the essential concepts of the computer solution.

Studies of human language attest to the importance of abstraction in communication, as illustrated in the following description by Damasio and Damasio in their article "Brain and Language" [31]:

> Language arose and persisted because it serves as a supremely efficient means of communication, especially for abstract concepts. . . . It helps to categorize the world and to reduce the complexity of conceptual structures to a manageable scale.
> The word "screwdriver," for example, stands for many representations of such an instrument, including visual descriptions of its operation and purpose, specific instances of its use, the feel of the tool or the hand movement that pertains to it. . . . The cognitive economies of language—its facility for pulling together many concepts under one symbol—make it possible for people to establish ever more complex concepts and use them to think at levels that would otherwise be impossible.

The C++ programming language provides many mechanisms for expressing abstractions and relationships among them. Some of these, like subroutines and data structures, are provided in languages widely used in scientific and engineering programming; some, like `virtual` functions and templates that we introduce in this book, are not commonly available in languages used in technical programs. No programming language comes close to the expressive power of natural languages. On the scale set by FORTRAN and C, however, C++ stands far ahead in the level of discourse it allows.

1.1 Object-Oriented Programming

The pivotal new mechanism for expressing abstractions directly in C++ is support for *object-oriented programming*. Three fundamental ideas characterize object-oriented programming: objects, class hierarchies, and polymorphism.

An *object* has state—data—and behavior—functions. Objects combine data that represent state with functions (subroutines) that access or modify the data. Objects generalize subroutines by adding data; the data in turn may influence the outcome of calling an object's functions. Each object is created from a *class*, a specification of the object's data and function. All objects of the same class have common operations but (generally) separate state.

Class hierarchies organize classes for reuse, with classes at the bottom inheriting operations from classes above: Operations need not be recoded for classes with common components. This mechanism is called *inheritance;* it extends the subroutine's packaging of common function to a packaging of common data structure and function.

Polymorphism allows different kinds of objects that share common behavior to be used in code that only requires that common behavior. While inheritance supports the use of 2D arrays to build matrices or tensors or images, polymorphism allows us to write one function to multiply two matrices, whether they are stored with rows adjacent, or with columns adjacent, or in a packed format.

Describing computations in terms of objects and classes brings important benefits:

Direct Expression. Objects are natural metaphors for both physical objects and abstract entities. Expressing computations in terms of objects reduces the gap between concept and program.

Malleability. Good programs evolve; evolution is easiest when the modifications are local. Objects combine data with functions to manipulate that data—allowing localization—and access to objects' data is restricted—enforcing localization.

Extensibility. Using inheritance, new objects and their behaviors can be defined as incremental modifications and extensions of existing objects.

Abstraction. Using polymorphism, similarities among objects can be expressed in the program, allowing us to write code in terms of the similarities without regard to the differences.

We hope this book will help you harness these benefits of object-oriented programming for your projects.

1.2 Why C++?

Among all object-oriented programming languages, we advocate C++ as a replacement for FORTRAN in engineering and scientific work. Here's why:

Availability. We program on personal computers, workstations, mainframes, and supercomputers: We need a language available on essentially all machines. C++ is widely available, because compilers are widely available and because of its wide popularity with systems programmers, the people who write compilers.

Portability. We move our programs between machines: We need an official standard, or at least a de facto language definition. Currently the book *The Annotated C++ Reference Manual* [44] is the de facto standard. Methodical improvements in the language by its designer, Bjarne Stroustrup [105, 107, 109], have essentially eliminated nonstandard language extensions. ANSI (American National Standards Institute) committee X3J16 and ISO (International Standards Organization) committee WG21 are working together toward an official standard. Meanwhile most C++ compiler vendors are tracking the decisions of the committees so that new features accepted by the committees are appearing in commercial compilers.

Efficiency. Despite dramatic improvements in the performance of computer hardware, there remain problems for which machine efficiency is important. We need a language that can be compiled into very efficient code when needed. C++ was designed from the ground up for efficiency comparable to C, the language of choice for computer operating systems and other systems demanding high efficiency.

Correctness. We want a language that will catch as many errors as possible as early in the programming process as possible. C++ can catch many errors during compilation and/or linking.

Generality. We work on many different types of problems with computer programs: We need a general-purpose language. C++ was not designed solely for one particular subfield of programming.

Libraries. We want to use the results of others when we can: We need a language that can interface with operating systems, graphics packages, custom hardware, and numerical packages. C++ programs can call C functions (subroutines), and through them (or in some implementations, directly), FORTRAN subroutines. We need not abandon our time-tested FORTRAN libraries: Instead we can build C++ interfaces that facilitate their use. New libraries, written in C++, are also available in many areas.

1.3 What About . . . ?

We investigated alternative languages before choosing C++. Each has some appeal, but no other language has the combination of dramatic new capabilities and practical usability available in C++. FORTRAN, C, and PASCAL are all practical—they are available, portable, and efficient—but their approach to programming is too concrete. The structural improvements of C and the type-checking improvements of PASCAL are not sufficient to dislodge FORTRAN as the most useful scientific programming language. Nonstandard extensions to FORTRAN-77 have improved FORTRAN in small ways, but have not removed its fundamental deficiencies.

The standard for a new version of FORTRAN, called FORTRAN-90 [78], has recently been published. It solves some of the problems with FORTRAN-77, provides direct support for array operations, and includes some of the features of C++, including a limited form of objects, but without inheritance or polymorphism. Thus far FORTRAN-90 has not been enthusiastically embraced by the FORTRAN community and compilers are not widely available.

Ada [53, 15] also provides a limited form of objects, again with neither inheritance nor polymorphism. Thus it provides the benefit of direct expression, but not the levels of malleability, extensibility, and abstraction available in languages with inheritance and polymorphism. On the other hand, Ada supports concurrency and synchronization, features not provided in C++.

Smalltalk [49] provides objects with both inheritance and polymorphism. It comes with a programming environment and an extensive library of classes for data structures and user interfaces. Its programming environment yields high productivity, and the language is well suited for rapid prototyping of ideas. On the other hand, many errors that C++ detects during compilation are not detected in Smalltalk until the program runs. Smalltalk programs are generally less efficient than C++ programs; in programs where this doesn't matter, Smalltalk may be suitable.

Objective-C is a language that "was formed by grafting the Smalltalk-80 style of object-oriented programming onto a C language rootstock" [29, p. 49]. It provides some of the flexibility of Smalltalk, but without the benefits of Smalltalk's programming environment; it has similar drawbacks. Objective-C has not found widespread acceptance and has a limited user community.

Eiffel [81] is another object-oriented language with both inheritance and polymorphism. It is closer in spirit to C++ than to Smalltalk. Having been designed without the goal of C compatibility, it has a certain elegance that C++ lacks; it also has features like assertions that aid programming with abstractions. However, it has not found widespread acceptance and has a limited user community.

APL [19, 45], Mathematica [120], Matlab [25], and similar programs with extensive programming capabilities make short work of many simpler problems in scientific computing. They supplement rather than compete with C++ as a tool: They succeed by sacrificing generality and usually efficiency or sophistication.

C++ is not the "perfect" programming language; its design aims were too practical for that [108, 109]. Some compromises in its design were made so that it could be a superset of the C language [68]; others were made in the name of efficiency. Some advanced techniques made possible by C++ would be more elegant in a language designed anew. Since C++ is not specialized for scientific and engineering work, useful scientific tools must be built or obtained from others. Finally, C++ presently relies on the same edit, compile, link, and run sequence we are familiar with from other languages. The more sophisticated facilities of C++ would benefit from improvements in this scheme.

If C++ is not the perfect language, it is far ahead of the FORTRAN, C, and PASCAL family. The differences among FORTRAN, C, PASCAL, and modern BASIC have little consequence compared to the differences between this group and C++. Like the other sophisticated tools we use in science and engineering, C++ is not easy to learn. That's why we wrote this book!

1.4 Program Design

To write a program, you must design it; to design a program, you must have written it. Without design, programs grow haphazardly, with new pieces added without regard for how they will work with the whole program. Without evidence and experience from implementation, design strays, specifying unimplementable and unnecessary code, or satisfying requirements that change before the code is written.

Computer scientists have been working on this design-versus-implement dilemma for some time. *Object-oriented software development*, coupled with the use of appropriate languages, like C++, provides part of the answer. Overall, object-oriented software development confronts corrections, modifications, and improvements as fundamental parts of the programming process, essential to producing quality software. The basic idea of object-oriented development is to decompose a software system on the basis of objects—entities characterized by actions—instead of on the basis of functions or data, as is done in more traditional design methodologies. The concept of an object aids in isolating programming components so that they can be more independently designed and implemented.

The intimate connection between design and programming in object-oriented languages is key to their success; it puts feedback into the design cycle, encouraging the discipline and experimentation that leads to high quality. Objects allow

programs to be designed in the same way that airplanes, houses, and computers are designed: The overall goal dictates the general relationship of the parts, but the exact specifications of each part depend on the specifications of its neighbors in ways that can be discovered only by iterating again and again through each subsystem. The design and the implementation of successful programs evolve together, each helping the other in incremental steps.

1.5 Organization of the Book

Our book is organized on the premise that object-oriented programming and program development are the key topics, and C++ is the vehicle for expressing and implementing our designs. We therefore scrimp to the extent possible on detailed discussions of language syntax and rules available in other books (see Notes and Comments 1.8 at the end of this chapter); marginal notes direct you to relevant portions of *The Annotated C++ Reference Manual* [44] for detailed information on syntax and rules. We emphasize general concepts, advanced techniques, and particular styles that will help you write object-oriented programs. Our examples are drawn from scientific and engineering applications, but the concepts, techniques, and styles are broadly applicable.

ARM §1.1

The book is organized into three parts. The first part is intended to give you a working knowledge of C++. The second part introduces object-oriented programming and design techniques, emphasizing the various ways to express commonality and abstraction. The third part illustrates advanced applications and techniques, primarily through examples.

After this chapter, Part I splits into two tracks: If you know FORTRAN, but not C, continue with Chapter 2; if you know C, continue with Chapter 3. Both chapters build on what you already know to cover the fundamental types and operations, control structures, simple input and output, and functions. The two tracks merge at Chapter 4, where we get our first glimpse of classes and objects. Chapter 5 concentrates on the mechanics of functions, including how they are declared and called, how arguments are passed and values returned, and how multiple functions with the same name but different arguments can be used. Chapter 6 then connects classes and functions, examining the essential role of functions in classes. Chapter 7 covers the basics of object lifetime and memory management. Part I concludes with an example in Chapter 8 that introduces you to issues and techniques in C++ program design.

Part II introduces groupings of classes into categories for the purpose of expressing various kinds of commonality. We focus on how to select objects, understand relationships among them, and express the relationships in C++. Chapter 9 covers interfaces for expressing common object behaviors. Chapter 10 describes

using inheritance to express common object implementation and the relationship between interfaces and implementation via inheritance. Chapter 11 explains how templates can be used to express common code structure, including how interfaces, inheritance, and templates can be combined. All of the techniques for expressing commonality exploit C++'s type system: Chapter 12 analyzes the interaction between categories of classes and types and it introduces a fourth category, classes sharing function-structure commonality.

Part III uses substantive examples both to reinforce the techniques introduced in Part II and to introduce new techniques. Since C++'s built-in arrays are inadequate for scientific and engineering programming, we introduce a system of array classes in Chapter 13. Chapter 14 develops pointer classes that can automate some of the memory management tasks that must be addressed in sophisticated C++ programs. Chapter 15 demonstrates the use of classes to provide an object-oriented wrapper for the FORTRAN-callable LAPACK subroutine library for linear algebra. Chapter 16 builds a set of classes that model abstract algebra (groups, rings, etc.) and uses those classes to implement classes for dimensional analysis. Chapter 17 illustrates techniques for manipulating functions as objects, showing how functions can be applied concisely to elements of collections like arrays. Chapter 18 discusses issues in calling C and FORTRAN functions from C++. The book concludes, in Chapter 19, with a data modeling problem; the solution uses many of the techniques developed in the book.

1.6 Source Code

All code in the book has been compiled and executed on an IBM RISC System / 6000 using the IBM C Set ++ for AIX/ 6000 compiler, version 2.1. Our code uses templates and exceptions, both relatively new features of C++. These features were described in the *Annotated C++ Reference Manual* [44] as "experimental." They have since been accepted by the standardization committees and supported in several (but not all) commercially available compilers. We expect most compilers to implement these features soon.

All of the source code that appears in the book can be obtained for noncommercial use from Addison Wesley Longman via http://www.awl.com/cseng/titles/0-201-53393-6. The code is stored in a "tar" file compressed with "compress" and called SciEng.tar.Z, and in a zipped file called SciEng.zip.

1.7 Contacting the Authors

We are eager to receive your comments, corrections, and suggestions. We can be reached via e-mail at C++SciEn@aw.com.

1.8 Notes and Comments

The computer science and trade literature discusses the issues and details of object-oriented programming using a variety of terminologies; none is universally accepted. Throughout the book we adopt the C++ names for these concepts. Occasionally we have introduced new terminology for concepts that we believe are not clearly labeled.

2 Object-oriented program design techniques are receiving increasing attention and even formalization. Bjarne Stroustrup, the designer of C++, gives his answer to the question "What is object-oriented programming?" in [106] and discusses object-oriented design in [107]. The paper by Booch [14] and the books by Booch [16], Cox [29], Meyer [81], Wirfs-Brock *et al.* [117], and Rumbaugh *et al.* [97] all discuss object-oriented program design. Meyer [80] discusses the relationship between object-oriented program design and code reusability.

3 Software quality—what it is and how to achieve it—is the goal of *software engineering*. Many of the early important papers in software engineering are collected in two books edited by Yourdon [121, 122]. Papers specifically related to software quality are collected in [23]. Many different program design methodologies have been proposed and used with varying degrees of success; see [12] for a survey of popular methodologies and [102] for a text.

4 Parnas [87, 88, 89] has long advocated techniques for program design that anticipate change, and many of his ideas have influenced object-oriented program design.

5 Structured programming advocates the use of stepwise refinement as a design technique. Although structured programming has been characterized as programming without go-to's, the real message is much more significant. See the classic papers of Dijkstra [37, 38, 39], Knuth [66], and Wirth [118, 119].

6 If you are familiar with some of the computer science literature on program design, you may ask how object-oriented program design relates to top-down and bottom-up design approaches. *Top-down* design roughly corresponds to, "Figure out what you need to do, and then figure out how you are going to get it done." Stated somewhat ungraciously, top-down design is structured programming scaled up to programming-in-the-large. The independent development of the components in a programming problem, the *bottom-up* design approach, is roughly, "Get all these connectable parts working, and then snap them together to make the big program work." Bottom-up design solves the problems of top-down design, but in its purest form it lacks the necessary direction and drive characteristic of top-down refinement. We end up wasting time on nicely designed modules that do not address the problems in the final program or that have to be altered extensively to fit the actual, rather than the expected, programming problem. C++ and object-oriented design provide a workable combination of these seemingly disparate methods.

7 Brooks argues persuasively in [18] that there is not now and there never will be a "silver bullet" that will make software costs drop as dramatically as hardware costs have. We agree, and we certainly do not offer C++ as that panacean "silver bullet."

1.8 The most detailed, concise, and up-to-date definition of C++ is *The Annotated C++ R erence Manual* [44], often abbreviated *ARM* after the title, but it is not intended as introduction. There are more than 100 books available about C++; see [2] for a li Lippman [74] and Horstmann [59] are good introductions to the language, and recommend either to you as a complement to our book. Stroustrup [107] provides more advanced treatment and discussion of design issues and techniques. Gorlen, O low, and Plexico [51] describe many of the advanced features of the language, in t context of one particular class library. Coplien [27] is an advanced treatment that e phasizes C++ idioms. Meyers [82] offers pithy guidelines for writing good C++ pr grams. Cargill [22] offers case studies to illustrate various issues in C++ programmin Stroustrup [109] describes the history of C++ and its evolution.

1.9 Exercises

1.1 Why do you write computer programs? Do you consider it difficult or easy? What fra tion of the work you invest in a program is creative? What fraction is tedious? Wh fraction highlights your special skills? What fraction could be done by a nonspeciali in your field?

1.2 How much of your programming effort is directed to completely new code? to co related to previous efforts? to revising and modifying existing code?

1.3 Which of the advantages listed in Section 1.2 would be most valuable to you?

1.4 FORTRAN has been around for more than 30 years. Why? Will scientific and enginee ing programs be written in FORTRAN 30 years from now?

CHAPTER 2

Basics for FORTRAN Programmers

In this chapter, we take a quick tour of C++'s basic features, leading you to the point at which you can read and write simple programs. We assume that you know FORTRAN, and we often compare C++ features with their FORTRAN analogs. If you are an experienced C programmer, read Chapter 3 instead; however, novice C programmers may benefit from reading both chapters, in order.

2.1 A First Program

A C++ program consists of *statements*, grouped into *functions*. A *declaration statement* declares the name and type of a variable or function and is roughly analogous to a FORTRAN explicit type statement (e.g., `INTEGER`, `REAL`). Other statements control program execution. Functions group declaration and other statements into named units that can be executed by calling the function and that typically reflect the logical organization of the program; C++ functions are analogous to FORTRAN `SUBROUTINE` and `FUNCTION` subprograms.

Statements are terminated by a semicolon and are grouped within braces ({ . . . }). Most statements contain *expressions*, sequences of operators, function calls, variables and constants that specify computation. Variable and function names are of arbitrary length and consist of upper- and lowercase letters, digits, and underscores; they may not start with a digit, and case is significant. All C++ keywords are written in lowercase.

C++ programs can be laid out in free form, with no particular meaning assigned to any column and no need to indicate continuation across lines. Indeed lines have no significance except in comments. Comments may appear anywhere. A comment is begun by the comment delimiter // and terminated by the end of a line, like this:

ch2/comments.C

```
a = b + c;     // This part is a comment.
```

Alternatively, a comment can be enclosed in comment brackets like this:

ch2/comments.C

```
/* This is a comment. */
```

The C++ compiler reads the program source code from one or more files. We'll sometimes need to know the names of files used in our examples, so we need to agree on some naming conventions. We'll assume that file names can have a suffix and that C++ source code is put in files with the suffix .C, .h, or .c, and that FORTRAN code is put in files with the suffix .f. If your system uses a different naming convention, adjust accordingly. The preceding comment examples illustrate a convention we use throughout this book: Each code fragment is annotated with the name of the file that contains it. (You can obtain these files; see Section 1.6.) When we want to intersperse explanatory prose with file contents, we simply continue the file later. You can find all code fragments contained in a file by looking for the file name in the Source File Index.

The most trivial C++ program looks like this:

ch2/trivial.C

```
int main() {
    return 0;
}
```

This program consists of a single function. It returns a value of type int, is named main(), and, as indicated by the empty parentheses, does not expect any arguments.

The name main is special in that a C++ program always executes the function called main. Most operating systems expect a program to return an integer status value when it terminates execution. In C++, that status value is the value returned by the function main. Therefore, main should be defined to return an integer value as we have done.

- Every C++ program must contain exactly one function named main. It should return an integer value.

This illustrates another convention: We often summarize important language rules and offer usage advice in text that is indented and set off with a square box.

The *function body* of main appears between opening and closing braces. Here the function body consists of a single return statement, which causes the function to finish executing and specifies a value to be returned.

Now let's write a C++ program that actually does something. Here is a program that reads and prints three floating point numbers:

ch2/regurgitate.C

```
#include <iostream.h>

int main() {
    // Read and print three floating point numbers
    float a, b, c;
    cin >> a >> b >> c;
    cout << a << ", " << b << ", " << c << endl;

    return 0;
}
```

As the compiler reads the file regurgitate.C, it immediately encounters a *preprocessor command*, a command that starts with a pound sign (#). This particular command, #include, causes the compiler to suspend reading from the current file (here regurgitate.C) to read the file specified (here iostream.h); when that file is read, the compiler resumes where it left off with the original file (regurgitate.C).

The file iostream.h is an example of a *header file* (hence the suffix .h). Header files typically contain declarations necessary for correct use of functions contained in function or class libraries (analogous to FORTRAN subroutine libraries). The header file iostream.h, which is supplied with C++, contains the declarations needed to use C++'s input and output (I/O) operations.

- Include iostream.h in any file that contains C++ input or output operations.

The first (noncomment) line of the function body declares three floating point variables a, b, and c. The keyword float is a type analogous to FORTRAN's REAL. The variables are not given any initial value.

The next statement reads the values from the input. The C++ input/output library defines cin and associates it with a standard source of input, usually the terminal. The >> operator extracts a value from an input source (cin) and puts the value into a variable; as shown, >> operators can be strung together to read more than one item. Specifically, three numbers are extracted from the input and put into a, b, and c. This kind of input is analogous to FORTRAN's list-directed READ statement.

The third line of the function body prints the values of the variables. The C++ input/output library defines cout and associates it with a standard output place, usually the terminal. The << operator inserts output into an output sink (cout). As shown, << operators can be strung together to write more than one item. The statement in our example writes the value of a, then a character string consisting of a comma and a space, then the value of b, and so on. However, in contrast to FORTRAN's WRITE statement, which always starts a new line of output, << never starts a new line unless directed to do so. The C++ input/output library provides

endl for controlling when new lines are begun: Writing endl ends the current line of output. More generally, input and output operations can be controlled by a variety of *manipulators*, like endl, provided by C++'s input/output library.

Think of a, b, and c as the coefficients of the line equation $ax + by + c = 0$. We now extend our program to compute and print the line's x- and y-intercepts:

ch2/intercept

```
#include <iostream.h>

int main() {
    // Read the coefficients of a line equation in the form ax+by+c = 0,
    // then print the line's x- and y-intercepts.

    // Read the equation coefficients.
    float a, b, c;
    cin >> a >> b >> c;

    // Print the equation coefficients.
    cout << "Coefficients: " << a << ", " << b << ", " << c << endl;

    // Compute and print the x-intercept.
    cout << "x-intercept: ";
    if (a != 0) {
        cout << -c / a << ", ";
    }
    else {
        cout << "none, ";
    }
```

The function begins with a comment that states the program's purpose and use. Such comments should be crisp and concise: They should provide sufficient detail for a reader to understand how to use the program, but not more.

The first statement declares the variables that hold the coefficients; the next statement reads their values. After printing the coefficients, the x-intercept is computed and printed. The if statement tests whether the value of a is nonzero. If it is not zero, an x-intercept exists and it is computed and printed by the code on the following line; otherwise the else clause is executed to print a message saying that no intercept exists. This is analogous to the FORTRAN block-IF statement:

ch2/ifthel.f

```
      IF (A .NE. 0) THEN
          PRINT *, -C/A, ', '
      ELSE
          PRINT *, 'NONE, '
      END IF
```

The y-intercept is computed and printed in like manner:

ch2/intercepts.C

```
    // Compute and print the y-intercept.
    cout << "y-intercept: ";
    if (b != 0) {
        cout << -c / b << endl;
    }
    else {
        cout << "none" << endl;
    }

    return 0;
}
```

The program completes by returning to the operating system. The value 0 is returned here because our operating system treats zero as meaning successful completion.

The details of compiling and running a C++ vary from system to system; consult the manual for your system for instructions. Running this program on the input 1 2 3 produces the following output:

```
Coefficients: 1, 2, 3
x-intercept: -3, y-intercept: -1.5
```

Many scientific and engineering programs need to call common mathematical functions such as sqrt() and cos(). FORTRAN provides intrinsic mathematical subroutines for such computations. In C++, the *standard math library* provides roughly equivalent functions. To make such functions available, you must include the header file math.h in your program and arrange (in a system-dependent way) for the subroutines to be linked with your program. For example, the following program reads an angle in degrees and prints its cosine:

ch2/cosang.C

```
// Read an angle in degrees and print its cosine.
#include <iostream.h>
#include <math.h>

int main() {
    float angle;                                    // Angle, in degrees
    cin >> angle;
    cout << cos(angle * M_PI / 180.0 ) << endl;     // M_PI is from <math.h>
    return 0;
}
```

2.2 Variables, Objects, and Types

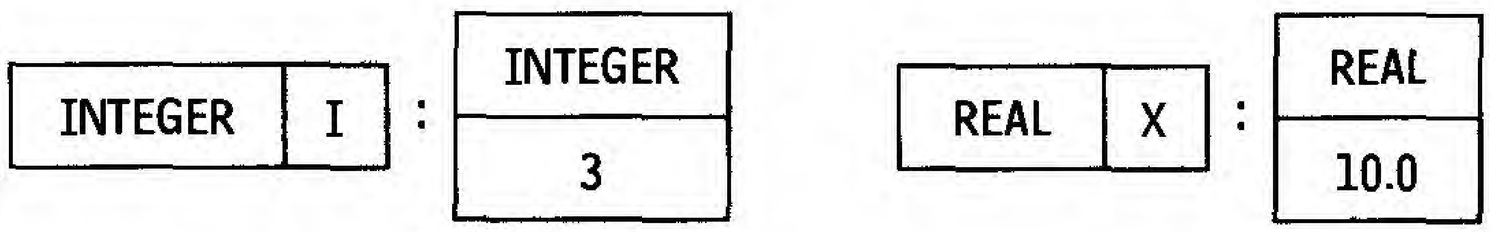
From a microscopic viewpoint, a computer program specifies a procedure for altering the values of objects, using symbolic references to the objects called variables. An *object* is an area of the computer's memory that has a *value*, a particular bit pattern, stored in it. The *type* of an object specifies how the bits stored as the object's value are to be manipulated. A *variable* is an association between a name and an object; the type of a variable specifies the type of objects that the variable can be associated with.

The FORTRAN code

ch2/simple.f

```
INTEGER I
REAL X
DATA I/3/, X/10.0/
```

defines two objects and gives them initial values. We illustrate the resulting variables, objects, and values like this:



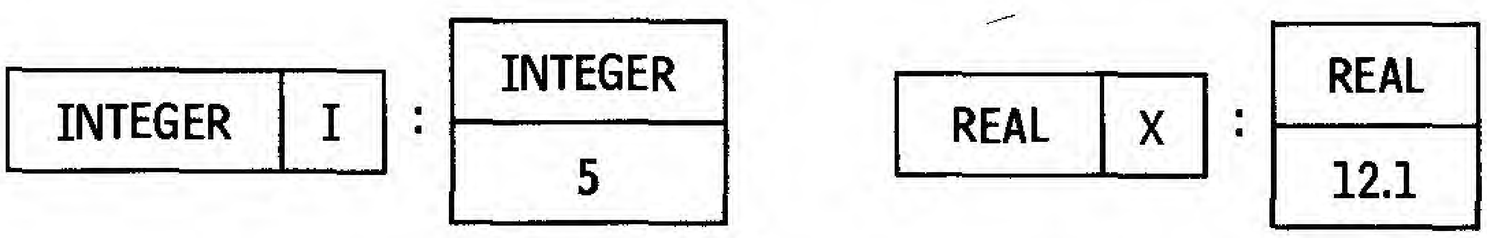
Each variable-object association is represented by two boxes separated by a colon: The variable is to the left and the object is to the right of the colon. Variable boxes are split into two parts giving the variable's type and name. Object boxes are also split into two parts, with the top part containing the object's type and the bottom part containing the object's value. In the preceding example, the variable named I has type INTEGER and is associated with an object of type INTEGER with value 3; the variable named X has type REAL and is associated with an object of type REAL having value 10.0.

Executing the FORTRAN code

ch2/simple.f

```
I = 5
X = 12.1
```

changes the values of the objects, giving



The assignment statements have changed the value of each object but not the types or the association between names and objects.

FORTRAN also allows new associations to be formed between variables and objects. Consider the code

ch2/simple.f

```
      SUBROUTINE S(A, B)
      REAL A, B
      A = B
*     ...
      END
```

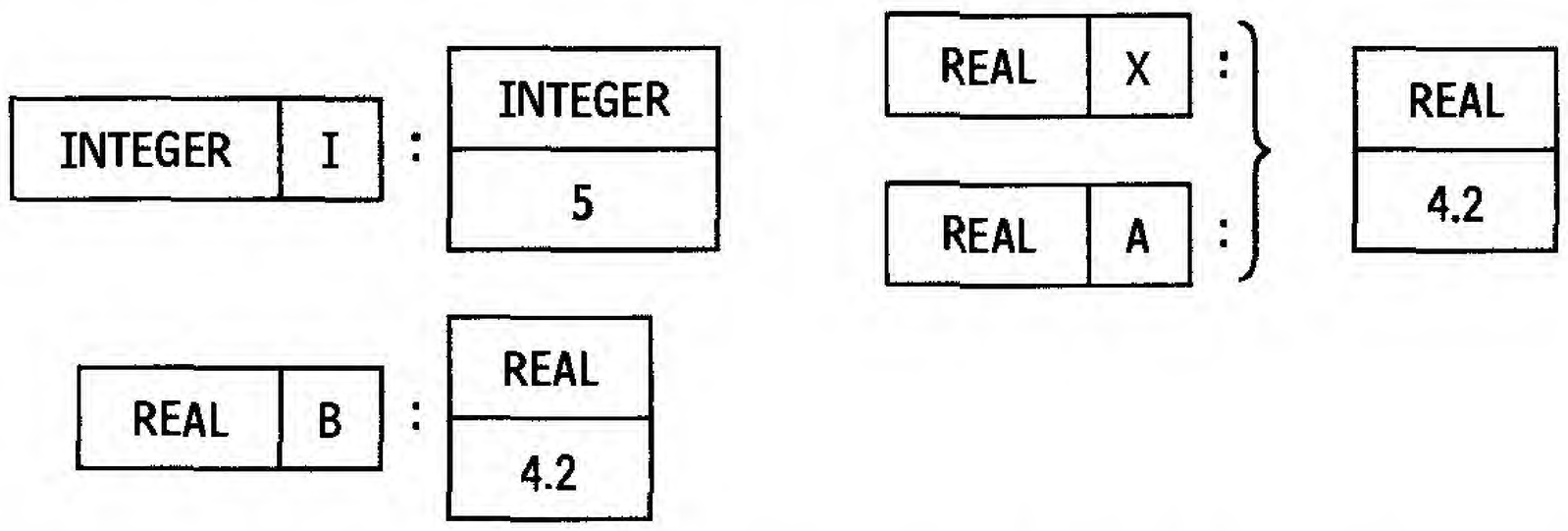
and add the following line to the previous example:

ch2/simple.f

```
      CALL S(X, 4.2)
```

Immediately before calling S, we have these objects:

There is an object with value 4.2, but it is not associated with any variable. After the assignment in S, but before the END, the objects and associations are



Passing an argument in FORTRAN creates an additional association between a name and an object without creating any additional objects. When the subroutine finishes, the name association is broken and x is left with the value 4.2.

Now let's look at analogous C++ code. The following C++ code declares two variables, as in the preceding FORTRAN example:

ch2/simplecpp.C

```
int i = 3;
float x = 10.0;
```

The int type is comparable to FORTRAN's INTEGER, and float compares to FORTRAN's REAL. Notice that upper- and lowercase are significant in C++ (e.g., X is

not the same as x), that initial values may be specified in the same statement that introduces the variable name, and that each statement ends in a semicolon. The C++ statements

ch2/simplecpp.C

```
i = 5;
x = 12.1;
```

change the values just as in the FORTRAN example.

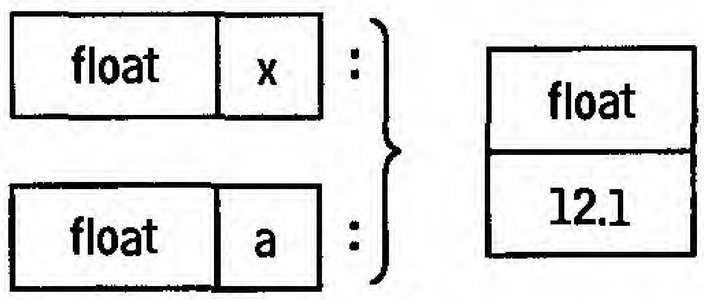
The most significant difference between this C++ example and its foregoing FORTRAN counterpart is that the declarations are mandatory in C++ but optional in FORTRAN. Each and every variable must be declared in C++, with its type given explicitly (the initial value is optional). There is no counterpart to FORTRAN's IMPLICIT typing system, which declares a variable's type based on the first letter of the variable's name; it is as if IMPLICIT NONE appeared in every FORTRAN subroutine. This burden of explicit declaration is repaid when the C++ compiler analyzes complicated code and detects type errors.

As in FORTRAN, C++ allows new associations to be created between variables and existing objects. The C++ mechanism is explicit and need not involve a subroutine call. The code

ch2/simplecpp.C

```
float x = 12.1;
float& a = x;
```

creates an association between the variable a, called a *reference*, and the object referred to by x. In pictures it creates



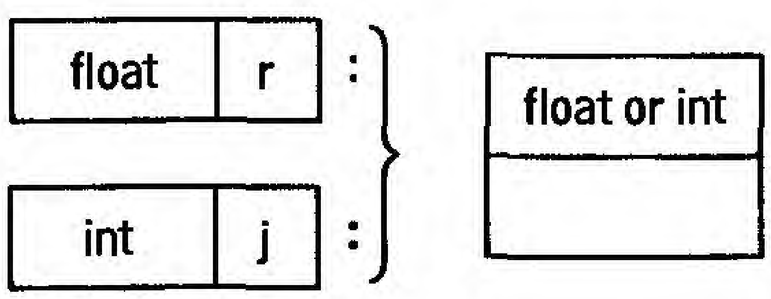
The variables x and a both access the same bits in the same way.

Finally, as with FORTRAN's EQUIVALENCE statement, two names and two different types can be associated with one object. The mechanism is called a union, as in

ch2/simplecpp.C

```
union { float r; int j; };
```

Both r and j access the same object using different interpretations of the bits. The corresponding diagram is



The union type is useful for reducing memory requirements when the valid type associated with an object can be established in each place the object is used. Union types are as dangerous and confusing as FORTRAN's EQUIVALENCE statements.

- Avoid union types.

2.3 C++ Fundamental Types and Operations

Objects in FORTRAN and C++ have *types*; while the number and meaning of the bits in an object are determined by the object's type, its type also defines the *operations* that can be used to manipulate the object. Giving objects type allows *type-checking rules* to detect erroneous operations on objects and *type-conversion rules* to specify what happens when an operation expects an object of one type but gets an object of another type. For example, FORTRAN has a type-checking rule that requires the operands of an addition operator (+) to be of the same numeric type. It also has a type conversion rule that defines the result of adding a REAL and a COMPLEX to be the result of first converting the REAL to a COMPLEX and then adding the two COMPLEX numbers.

C++ provides built-in (i.e., predefined) types, operations, type-checking rules, and type conversion rules that are analogous to those of FORTRAN. The built-in types are described in the remainder of this chapter, beginning in this section with the fundamental types. More advanced aspects of C++'s type system with no FORTRAN analog, including the ability to define new types, are described beginning in Chapter 4.

2.3.1 Numbers

Built-in numbers in C++ are either integer or floating point, corresponding roughly to FORTRAN's INTEGER and REAL. Both integer and floating point numbers are available in several precisions, with the correspondence between FORTRAN and C++ numbers shown in Table 2.1. The char type is listed with the other integer types because a char is defined as an integer with enough precision to hold a character. C++ does not provide complex numbers as a built-in type, but most C++ compilers come with the definition of a programmer-defined complex type that provides all of the usual operations and functions. ARM §

Both integer and floating point constants are written as you would expect from FORTRAN. One warning, however: ARM §

- An integer constant that begins with 0 (zero) is interpreted to be in octal (base 8); beware: 020 and 20 are different!

The exact numerical properties of numbers are determined by the underlying

C++ type	*Size (bytes)*	*FORTRAN analog*
char	1	CHARACTER*1
short int	2	INTEGER*2
int	2 or 4	INTEGER*2 or INTEGER*4
long int	4 or 8	INTEGER*4 or INTEGER*8
float	4	REAL*4
double	8	REAL*8
long double	16	REAL*16

Table 2.1 Numeric Types and Typical Sizes. The actual size of each object is implementation dependent. Usually, int is the natural machine word size and either short or long int will be the same size as int. The type long int can be written long, and short int can be written short.

§3.2c hardware; each C++ implementation provides the header file limits.h, which defines the largest and smallest value for each of the integral types, and float.h, which defines the characteristics of floating types including their precision, maximum and minimum representable values, and exponent range.

2.3.2 Built-in Arithmetic Operators

C++ provides the arithmetic operators shown in Table 2.2. There is no C++ exponentiation operator, but exponentiation is provided by the pow() function in the standard math library. We miss FORTRAN's convenient exponentiation operator, but with this exception C++ has a richer set of operators. The majority of exponentiations in FORTRAN programs are squares; these can be coded as repeated multiplications or by calling a sqr() function, which can be implemented to be as efficient as repeated multiplication. Our implementation of sqr() appears in Section 4.4.

The increment and decrement operators provide a convenient shorthand for adding or subtracting 1 from a variable. Used as a prefix operator, such as ++i, the variable value is incremented (decremented) before the variable value is used; used as a postfix operator, such as i++, the variable value is incremented (decremented) after the value is used. For example, the program

ch2/prepostfix.C

```
int i = 1;
cout << i << ", ";          // Print i
cout << (++i) << ", ";      // Increment i, then print its new value
cout << i << ", ";          // Print i
cout << (i++) << ", ";      // Print i's old value, then increment i
cout << i << endl;          // Print i
```

C++	*Purpose*	*FORTRAN*
`x ++`	Postincrement	
`++ x`	Preincrement	
`x --`	Postdecrement	
`--x`	Predecrement	
`+ x`	Unary plus	`+ X`
`- x`	Unary minus	`- X`
`x * y`	Multiply	`X * Y`
`x / y`	Divide	`X / Y`
`x % y`	Modulus	`MOD (X,Y)`
`x + y`	Add	`X + Y`
`x - y`	Subtract	`X - Y`
`pow(x,y)`	Exponentiation	`X ** Y`

Table 2.2 Arithmetic Operators

produces the output

```
1, 2, 2, 2, 3
```

when it is run.

If the operands of a binary arithmetic operator have different type, one of the operands is converted to the type of the other and the result of the operation has that common type. This occurs so frequently that the applicable rules are called the *usual arithmetic conversions*. In simplified form, these rules boil down to the following: If both operands are of the same type, that type is the type of the result; otherwise both operands are converted to a common type. The return type of the operand becomes that common type. This common type is the type of the operand that appears earliest in the list `long double`, `double`, `float`, `long int`, and `int`. Operands of type `char` and `short int` are treated as if they were of type `int`. ARM

The usual arithmetic conversions usually do what you want, but not always: `(3 / 2)` is the `int` `1`, not the `double` `1.5`.

- *Warning*: If you intend a constant to be a floating point number, write it like a floating point number: not `4`, but `4.0`.

Numeric comparisons are provided by the relational operators shown in Table 2.3. They compare their operands and have the value `1` if the relation is true, and `0` otherwise. The usual arithmetic conversions apply. ARM

- *Warning*: The equality test operator (`==`) is distinct from the assignment operator (`=`).

C++	*Purpose*	*FORTRAN*
`x < y`	Less than	`X .LT. Y`
`x <= y`	Less than or equal	`X .LE. Y`
`x > y`	Greater than	`X .GT. Y`
`x >= y`	Greater than or equal	`X .GE. Y`
`x == y`	Equal	`X .EQ. Y`
`x != y`	Not equal	`X .NE. Y`

Table 2.3 Relational Operators

2.3.3 Logical Values and Operators

Unlike FORTRAN, C++ has no separate `LOGICAL` type. Instead logical values are represented by integers: Zero is false and nonzero is true. C++'s logical operators are shown in Table 2.4. For example, the `!` operator ("not") takes a numeric operand and yields the `int` `1` for a zero operand and the `int` `0` otherwise. The `&&` operator yields `1` if both of its operands are nonzero and `0` otherwise; the `||` operator yields `1` if either of its operands are nonzero and `0` otherwise. Both `&&` and `||` evaluate their left operand first and don't evaluate their right operand if they don't have to. For example, if `y` is zero, the expression `y && x/y` yields `0` and does not divide by `0`.

2.3.4 Characters

Constants of type `char` are written as a character enclosed in single quotes, like `'x'`. Characters with no printable representation can be entered by the *escape sequences* shown in Table 2.5. For example, the newline character is typed `'\n'`.

When used as operands of arithmetic and relational operators, `char` objects are converted to integers. For example, the statement

C++	*Purpose*	*FORTRAN*
`0`	False value	`.FALSE.`
nonzero	True value	`.TRUE.`
`! x`	Logical negation	`.NOT.X`
`x && y`	Logical and	`X .AND. Y`
`x || y`	Logical inclusive or	`X .OR. Y`

Table 2.4 Logical Values and Operators

C++ Escape Sequence	*Meaning*
\n	Newline
\t	Horizontal tab
\v	Vertical tab
\b	Backspace
\r	Carriage return
\f	Form feed
\a	Alert
\\	Backslash
\?	Question mark
\'	Single quote
\"	Double quote
ddd	Octal number with 1, 2, or 3 digits, *d*
\x*ddd*	Hexadecimal number with any number of digits, *d*

Table 2.5 Character Escape Sequences

ch2/charcomp.C

```
cout << ('a' < 'b') << " " << ('1' >= '3') << " " << ('a' != 'A') <<
    " " << ('a' == 'a') << " " << ('z' > 'y') << endl;
```

prints

```
10111
```

when run. Don't take the statement "chars behave like integers" too far, however, because the correspondence between characters and integers depends on the character set chosen by the compiler implementer. The ASCII character set is most common, but the EBCDIC character set is also used. With either of these character sets, testing equality, comparing letters of the same case, and comparing digits (0–9) will all work as expected. The standard header file ctype.h defines functions such as isupper(), isdigit(), and ispunct() for classifying chars. See, for example, [58, Chapter 12].

Character strings are provided by arrays of char elements. A sequence of characters enclosed in double quotes, like "abcd", are character array constants. We discuss these in Section 2.14.

2.3.5 Enumerations

C++ provides a way to define mnemonic names for integer codes grouped into sets. An *enumeration* is an ordered set of names: **ARM §7.**

ch2/enum.C
```
enum Color { red, orange, yellow, green, blue, indigo, violet };
```

This declaration defines a new type, Color, and seven unalterable variables of that type. For example,

ch2/enum.C
```
Color c = green;
```

declares the variable c to be of type Color with initial value green. C++ initializes each name in the enumerator list with an integer number code, starting with 0 and increasing by 1 going left to right. You may specify the integer values to be associated with each identifier in an enumeration, like this:

ch2/enum.C
```
enum Polygon { triangle = 3, quadrilateral = 4, pentagon = 5 };
```

Enumeration values can convert to ints when they are used as arithmetic and relational operands. For example, the expression c <= yellow yields 1 when c has any of the values red, orange, or yellow, and 0 otherwise. But an int is not convertible to an enumeration type: In Polygon p = triangle + 1, the sum of triangle and 1 gives the int 4 legally, but the int 4 cannot be assigned to p.

2.3.6 Bitwise Operators

Sometimes it is appropriate to manipulate an object's value as a bit pattern. For example, some experimental equipment attached to a personal computer can be controlled by reading and writing certain bit patterns through an 8-bit parallel interface. C++ allows the integral types (char and all lengths of int) to be manipulated as bit patterns using the operators shown in Table 2.6. The code

ch2/bitwise.C
```
char command;
enum commandFlags {enable = 0x01, pulse = 0x02, tone = 0x04, reset = 0x08 };
command = (enable | tone);
```

sets the value of command to the bits 00000101. A sequence of digits preceded by 0x is an integer constant specified in hexadecimal notation.

2.3.7 Assignment Operators

An *assignment operator* alters an object's value without altering its type or other attributes. The usual assignment operator (=) copies the value of its right operand into the object that is the left operand, applying the usual arithmetic conversions when necessary. C++ also provides the following *updating assignment operators*:

```
+=  -=  *=  /=  %=  >>=  <<=  &=  ^=  |=
```

C++	*Purpose*	*FORTRAN*
~i	Bitwise complement	NOT(I)
i & j	Bitwise and	IAND(I, J)
i ^ j	Bitwise exclusive or	IEOR(I, J)
i | j	Bitwise inclusive or	IOR(I, J)
i << n	Bitwise shift left	ISHFT(I, N)
i >> n	Bitwise shift right	ISHFT(I, -N)

Table 2.6 Bitwise Operators

In each case, for a variable a of a built-in type, a *op*= expr is equivalent to a = a *op* (expr). For example, a += 5 is equivalent to a = a+5.

2.4 Input and Output

In addition to arithmetic and bit-manipulation expressions, C++ uses operators for its basic input/output facility. Section 2.1 illustrates using the C++ iostream I/O facility to read and write simple strings and numbers. This section describes some additional capabilities that are commonly needed and some aspects of compatibility with FORTRAN I/O.

The iostream facility is based on the notion of *streams*. A stream is a sequence of bytes that are produced (output) or consumed (input) by a program. Usually a stream is associated with a file or a device, such as a terminal; in some systems, a stream can also be associated with a pipe, which is a mechanism for communication among different programs. Regardless, a stream is a place either to get or put bytes, and the C++ iostream system is a way of moving external data to and from objects.

The three streams cin, cout, and cerr, corresponding to the program's standard input, standard output, and standard error as defined by the operating system, are available automatically. On most systems, input from cin comes from the keyboard and output to cout and cerr goes to the screen. Operating system facilities can associate standard input, standard output, and standard error to, for example, files or other programs.

As data are read from an input stream, the bytes in the stream are interpreted according to the kind of input that is requested. The >> operator having a stream as its left operand requests that the bytes in the stream be interpreted as characters. If the variable x is a number, then the expression

ch2/iodemo.C

```
cin >> x;
```

reads characters from cin, converts them into a number of the appropriate type, and stores it into x. Blanks, newline characters, and tab characters, all called *white-space*, separate data items read by the >> operator; whitespace is otherwise ignored. For convenience, >> operators can be concatenated to read multiple data items:

ch2/iodemo.C

```
cin >> x >> y;
```

Output is done with the << operator:

ch2/iodemo.C

```
cout << x;
```

This causes the value of the variable x to be formatted into a sequence of characters and written to cout. No whitespace is inserted, so

ch2/iodemo.C

```
int i = 1;
int j = 2;
cout << i << j;
```

writes 12, probably not what you intended.

- Insert your own whitespace on output.

Whitespace can be inserted by writing character strings to the output. The previous example is correctly written as:

ch2/iodemo.C

```
int i = 1;
int j = 2;
cout << i << " " << j;
```

which writes 1 2. You will probably also want to split your output into multiple lines. A new line can be begun by writing the endl manipulator, which ends a line:

ch2/iodemo.C

```
int i = 1;
int j = 2;
cout << i << endl << j << endl;
```

This produces the output

```
1
2
```

Many FORTRAN programs expect input that is organized into fixed columns; most C++ programs do not produce fixed-column output. Hence writing a C++ program that produces output for later consumption by a FORTRAN program requires altering the behavior of C++ streams. In particular, C++ uses a lowercase e in exponential notation, and it omits the decimal point in floating point output if the number has an integral value (e.g., 20.0 is written as 20). This can be changed.

For example, here is a C++ program that writes three numbers that can be read by a FORTRAN program with FORMAT(2G15.8,F10.3):

ch2/fortran-compatible-io.C

```
#include <iostream.h>
#include <iomanip.h>

int main() {
    double a = 3.14159;
    double b = 1 / a;
    double c = 10 * a;

    // Use FORTRAN compatibility output.
    cout << setiosflags(ios::showpoint | ios::uppercase);

    // Write data in G15.8 format.
    cout << setw(15) << setprecision(8) << a;
    cout << setw(15) << setprecision(8) << b;

    // Write in F10.3 format
    cout << setiosflags(ios::fixed);
    cout << setw(10) << setprecision(3) << c << endl;

    return 0;
}
```

The formatting behavior of cout is set with three manipulators: setw for field width, setprecision for field precision, and setiosflags for various control flags. Inserting setw and setprecision into cout with the << operator affects the formatting of the next item inserted into cout. The manipulator setiosflags takes various flags that determine, for example, whether a decimal point is always shown (ios::showpoint) and the case of the E in scientific notation (ios::uppercase). To write in fixed format (i.e., F instead of G), we again set a flag: ios::fixed. To use these manipulators, the header file iomanip.h is required in addition to iostream.h. Other manipulators are available for controlling formatting; refer to your system's documentation or to [111] for details.

I/O to files is also accomplished via streams in C++. An output stream is connected to a file using an object of type ofstream, like this:

ch2/io.C

```
#include <iostream.h>
#include <fstream.h>

ofstream out("pi.out");
out << 3.14159 << endl;
```

The variable out is of type ofstream ("output file stream") and can be used just like cout. It is connected to the file named pi.out.

Similarly, input file streams are connected to files via objects of type ifstream, and files used for both reading and writing are objects of type fstream. To use these types, you must include the header fstream.h.

Finally, C++ also provides an analog of FORTRAN's internal file for converting numerical data to its text representation and vice versa, again via streams. For details, refer to [111].

2.5 Operator Precedence and Associativity

With some idea about basic types and their operations, we can consider the control of program execution. Programs execute operations on objects in a sequence specified by the programmer. In C++, as in FORTRAN, this sequence is specified in two levels—within expressions and across statements. Within an expression, the sequence is determined by operator precedence and associativity; these are the topics of this section. Across statements, the sequence is determined by *control structures*, which are the topics of Sections 2.6 and 2.7.

In mathematical notation, certain conventions tell us how to interpret equations: $ax + by + c$ is understood to mean "multiply a and x, multiply b and y, add the two products together, and add the resulting sum to c." We interpret the FORTRAN expression `A*X+B*Y+C` analogously.

The interpretation of any C++ expression is determined by the precedence and associativity of the operators in the expression. Each C++ operator has a precedence: Operators in an expression are evaluated in order of highest to lowest precedence. For example, multiplication and division have higher precedence than addition and subtraction: `a*x+b*y+c` has the expected meaning. The evaluation order for operators with equal precedence is determined by their associativity: left to right or right to left. C++'s arithmetic operators group left to right, so `a-b-c` is the analog of $(a - b) - c$ in math. Also as in math, parentheses in C++ override operator precedence and associativity: `a-(b-c)` overrides C++'s associativity rules.

C++'s operator precedence and associativity rules for arithmetic operators correspond to our expectations from mathematics. But most of us don't have any expectations about bitwise shift operators. For such operators, you should either use parentheses or refer to Table 2.7. (Some of the operators listed have not been discussed yet.) And remember, if you have doubts about the precedence or associativity of an operator, so might someone else reading your code.

- Use parentheses and spacing to make expressions easier to read.

C++ Operator	*Purpose*	*Associativity*	*Precedence*
::	Scope (unary)	Right to left	Highest
::	Scope (binary)	Left to right	
-> .	Member selection	Left to right	
[ ]	Subscripting	Left to right	
()	Function call	Left to right	
++	Postincrement	Left to right	
--	Postdecrement	Left to right	
sizeof	Object size	Right to left	
++	Preincrement	Right to left	
--	Predecrement	Right to left	
* & + - ! ~	Unary operator	Right to left	
new	Create object	Right to left	
delete	Delete object	Right to left	
()	Type cast	Right to left	
->* .*	Pointer to member	Left to right	
* / %	Multiplicative operators	Left to right	
+ -	Additive operators	Left to right	
<< >>	Bitwise shift operators	Left to right	
< > <= >=	Relational operators	Left to right	
== !=	Equality operators	Left to right	
&	Bitwise and	Left to right	
^	Bitwise exclusive or	Left to right	
|	Bitwise inclusive or	Left to right	
&&	Logical and	Left to right	
||	Logical or	Left to right	
?:	Conditional operator	Right to left	
= *= /= += -= >>= &= ^= |= %= <<=	Assignment operators	Right to left	
,	Comma operator	Left to right	Lowest

Table 2.7 Operator Precedence and Associativity. Operators are listed in order of decreasing precedence. Those operators between the same horizontal lines have the same precedence.

2.6 if Statements

The C++ if statement is analogous to FORTRAN's block-IF statement. Suppose that an experiment control program takes several actions if a temperature becomes dangerously high. In FORTRAN you might write

ch2/tempct.f

```
IF (TEMP .GT. MAXSAF) THEN
    PRINT *, 'EMERGENCY: Too hot--flushing'
    CALL FLUSH
END IF
```

Analogous C++ code looks like this:

ch2/tempctrl.C

```
if (current_temp > maximum_safe_temp) {
    cerr << "EMERGENCY: Too hot--flushing" << endl;
    flushWithWater();
}
```

The braces group the two statements so that both or neither are executed, like the `THEN` and `END IF` in the FORTRAN version. A sequence of any number of statements enclosed in braces is called a *block* and can be used wherever a statement is needed.

Formatting enhances code readability:

- Select a code formatting style and use it consistently.

We indent each line of a block by the same amount, place the opening brace at the end of a line, and place the closing brace on its own line, indented to align with the block-opening line. The details of code layout are less important than consistency; see Notes and Comments 2.1.

The general syntax of an if statement is

```
if (expression) {
    statement;    // statements in braces form a block
    ...
}
```

The *expression* must be in parentheses and must yield a numeric value. The statements in the block following the if and its expression are executed if the value is nonzero. The if may be followed by a single statement rather than a block:

ch2/simplecpp.C

```
if (x < 0) x = -x;    // x = abs(x)
```

However, we recommend the following:

- Use a block for an if statement body; exceptions should always be coded on one line.

Otherwise you might change

ch2/tempctrl.C

```
if (current_temp > maximum_safe_temp)
    cerr << "EMERGENCY: Too hot--flushing" << endl;
```

to

ch2/tempctrl.C

```
// WRONG!
if (current_temp > maximum_safe_temp)
    cerr << "EMERGENCY: Too hot--flushing" << endl;
    flushWithWater();
```

which is correct C++, but douses the equipment regardless of the temperature.

C++ also has a *conditional operator*, which returns one of two values based on a logical test:

ch2/simplecpp.C

```
x = (x < 0) ? -x : x;  // x = abs(x)
```

It has the form

expression ? *true-value* : *false-value*;

and it can be read, "if *expression* then *true-value* else *false-value*." Unlike if statements, the conditional operator always returns a value and always has an "else" part. The conditional operator notation is so compact that unreadable code can result.

- Use the conditional operator only for simple tests that fit on one line.

We can use the if–else statement to control the temperature before it becomes dangerously high. The if–else statement has the following general form:

```
if (expression) {
    block-1-contents;
}
else {
    block-2-contents;
}
```

If *expression* is nonzero, *block-1-contents* is executed; otherwise *block-2-contents* is executed.

A (still primitive) control strategy might be implemented like this:

ch2/tempctrl.C

```
if (current_temp > maximum_safe_temp) {
    // Emergency cool down.
    cerr << "EMERGENCY: Too hot--flushing" << endl;
    flushWithWater();
}
```

```
    else {
        // Normal control strategy.
        if (current_temp > operating_temp + temp_tolerance) {
            heaterOff();
            if (current_temp > operating_temp + 2*temp_tolerance) {
                coolingWaterOn();
            }
        }
        if (current_temp < operating_temp - temp_tolerance) {
            coolingWaterOff();
            if (current_temp < operating_temp - 2*temp_tolerance) {
                heaterOn();
            }
        }
}
```

The outermost if–else statement selects either the "emergency cool down" action or the "normal control strategy," but not both. Within the normal control strategy, the if statements are used to determine whether actions should be taken to lower or to increase the temperature; no action is taken if the temperature is within temp_tolerance of the desired operating temperature.

2.7 Loops

C++ has three kinds of loops, which are summarized and compared to equivalent FORTRAN code in Table 2.8. The while and do–while loops have no direct FORTRAN analogs but can be implemented in FORTRAN using IF and GOTO statements; the for loop is a generalization of the FORTRAN DO loop.

2.7.1 while Loop

The while loop

```
while (expression) {
    statement;    // while loop body
    ...
}
```

executes the loop body repeatedly while *expression* is nonzero (true). The loop-body could be a single statement not enclosed in the block's braces. The test is done before executing the loop body, which means that the loop body is executed zero or more times.

C++	FORTRAN
while (*expr*) {	10 IF (.NOT. *expr*) GOTO 20
...	...
	GOTO 10
}	20 CONTINUE
do {	10 CONTINUE
...	...
} while (*expr*);	IF (*expr*) GOTO 10
	init-stmt
	10 IF (.NOT.(*cont-expr*)) GOTO 20
for (*init-stmt; cont-expr; incr-expr*) {	...
...	*incr-expr*
}	GOTO 10
	20 CONTINUE
for (i=1; i <=j; i += k) {	DO 10 I=1, J, K
...	...
}	10 CONTINUE

Table 2.8 Summary of C++ and FORTRAN Loops. These are equivalent, but note that C++ arrays have index 0 as the first element and therefore most C++ loops begin with element 0.

- Use a while loop when it is possible that the loop body should not be executed.

This property makes the while loop suitable for reading and processing the contents of a file of unknown size. For example, the following code fragment reads numbers and prints a table of their square roots:

ch2/sqrtTable.C

```
float x;
while (cin >> x) {
    cout << setw(25) << x << setw(25) << sqrt(x) << endl;
}
```

The expression cin >> x sets x from a value read from cin; the expression is nonzero if the read succeeds and zero otherwise. Thus the loop stops at the end of the input file. The output statement is executed exactly once per input item, even if there are no input items (in which case the output statement is not executed). The manipulator setw sets the width of the next output field (see Section 2.4).

2.7.2 do–while Loop

The do loop

```
do {
    statement;    // do–while loop body
    ...
} while (expression);
```

executes the statements in the loop body repeatedly while *expression* is nonzero (true). The test is done after executing the loop body, which means that the loop body is executed at least once.

- Use a do loop when the loop body should always be executed at least once.

The do loop is convenient when the loop body computes a value needed as part of the termination condition. For example, a root of a nonlinear function $f(x)$ can be found by Newton's method [93, Section 9.4], which computes a sequence of numbers x_i that converge to a solution of $f(x) = 0$:

$$x_{i+1} = x_i - \frac{f(x)}{f'(x)}.$$

The iterative process terminates when $|x_{i+1} - x_i|$ is small enough. Letting the variable x hold the value of x_i and the variable dx hold the value of $|x_{i+1} - x_i|$, Newton's method can be implemented by the following do–while loop:

ch2/Newton.C

```
x = initial_guess;
do {
    dx = f(x) / fprime(x);
    x -= dx;
} while (abs(dx) > desired_accuracy);
```

Recall that `x -= dx` is equivalent to `x = x - dx`.

2.7.3 for Loop

The for loop combines loop initialization, termination condition, and incrementing into one construct with the general form

```
for (init-statement; continue-expression; increment-expression) {
    statement;    // for loop body
    ...
}
```

This for loop is equivalent to

```
init-statement;
while (continue-expression) {
    statement;    // for loop body
    ...
    increment-expression;
}
```

but is often more convenient. The order of the expressions in the for loop is analogous to the order in the FORTRAN DO loop—initialization, termination condition, and increment (cf. Table 2.8)—but in C++'s for loop the expressions are not limited to initializing, testing, and incrementing integers.

The *init-statement* in a for loop can be a declaration statement:

ch2/simplecpp.C

```
for (int i = 1; i <= 10; i++) {
    // Loop body
}
```

This declares and initializes the variable i to 1 and then executes the loop body until i exceeds 10, incrementing i each time at the end of the loop; if the loop body doesn't alter i, the loop body is executed 10 times. Writing the *increment-expression* as i++, instead of as i = i + 1, clearly indicates incrementing and is conventional C++ usage.

2.7.4 break and continue Statements

The break statement terminates the execution of the smallest enclosing while loop, do–while loop, or for loop. For example,

ch2/break.C

```
int x;
while (cin >> x) {
    if (x < 0) {
        cout << "Negative number " << x << " read" << endl;
        break;
    }
    // ... processing of input goes here
}
```

reads numbers from cin, testing that each one is nonnegative before processing it. If a negative number is read, a message is printed and the loop is terminated by execution of the break statement. Similarly, the continue statement skips the remaining code in the loop, but only for the current iteration.

- Use break and continue sparingly and only when their meaning will be clear.

If overused they lead to code similar to that created with many GOTO statements.

2.8 Declarations

C++ variables are created by *declaration* statements. A declaration is a statement like

ch2/simplecp

```
int i = 0;
```

The first word gives the type; it is followed by the variable name and an optional initial value following an equals sign. The type, int in this example, is one kind of attribute; the variable name (the i) is called a declarator; and the specification of the initial value (= 0) is called an initializer. In general, a declaration consists of four components:

allocator attributes declarator initializer;

The allocator and the initializer are optional. Some example declarations are shown in Table 2.9. Each component relates to various aspects of creating a variable and its associated object, as follows:

Allocator. The allocator specifies both where the object is to be located in memory and, to some extent, what part of the program can access the object.

Attributes. The attributes specify the type of the object being created and, optionally, that the object is a constant (described later).

Declarator. The declarator specifies the name of the variable being created. C++ names can be of arbitrary length and may have both upper- and lowercase letters, with case significant. Optionally, the name can be followed by (1) a positive integer between square brackets (e.g., [5]) which specifies that the object being named is to be an array; or by (2) an argument list enclosed in parentheses, which specifies that the object being named is to be a function. Arrays are introduced in Section 2.9 and functions in Section 2.16.

Declaration	*Attributes*	*Declarator*	*Initializer*
int x;	int	x	
int x = 3;	int	x	= 3
float y = 1.1;	float	y	= 1.1
int a[3];	int	a[3]	
const float e = 2.71828;	const, float	e	= 2.71828

Table 2.9 Example Declarations and Their Meanings

Initializer. The initializer specifies an initial value for the object being created. Its general form is an equals sign (=) followed by an expression that computes the initial value.

Since declarations are statements, they can be intermixed with other statements: They need not—and should not—be grouped together and placed before the executable statements, as is required in FORTRAN. We suggest that you

- Use one declaration statement for each variable; declare each variable just before its first use.

One-variable declarations are easier to read and edit than multiple-variable declarations; declaring variables near their use makes their type easier to find and edit.

2.8.1 const Declarations

Formulas that contain physical constants, such as Planck's constant, are usually written as a symbol, not as a number. This helps us recognize the physical significance of the formula, takes less time to write, avoids transcription errors, and allows for the occasional change in a physical constant. Similarly, C++ *constant objects* are symbols for objects with fixed value. In FORTRAN, such constants are specified with the PARAMETER statement:

ch2/simple.f

```
PARAMETER (H=6.6256E-34)    ! Planck's constant (mks units)
```

To declare a C++ variable to be constant, use the const type modifier just preceding the type name:

ch2/const.C

```
const float h = 6.6256e-34;    // Planck's constant (mks units)
```

This declares h to be an immutable floating point constant. The symbol h can now appear in computations or be printed, but its value can't be changed. (Despite the common use in physics of single character symbols for physical constants, it is a bad idea to use such a symbol as the name of a constant that is known to large parts of a program. Thus h_Planck would be a better choice for this constant.)

An arbitrary expression can be used to initialize a const variable. The word constant does not mean that the value has to be known before the program is run. Rather it means that once the variable is initialized, it can't be changed. The following code is correct:

ch2/const.C

```
void f() {
    float x;
    cin >> x;    // Read value of x
    const float xc = x;
    // ...
}
```

2.8.2 typedef Declarations

The C++ typedef declaration gives an additional name to an existing type:

ch2/typedef.C

```
typedef float distance;
```

This makes distance a synonym for float but does not define a new type and does not add any additional type-checking or conversion rules. For example,

ch2/typedef.C

```
typedef float force;
distance d = 10;
force f;
f = d;
```

declares the variables d and f to be of seemingly different types, but the assignment is allowed because both variables are really of type float.

The most important use of typedef is to plan for change. A program running on a personal computer with limited memory might handle small problems using single precision floating point and 16-bit integer arithmetic. The same program running on a workstation might solve larger problems using double precision floating point and 32-bit integers. If you declare integers to be of type intNumber instead of int, and floating point numbers to be of type floatNumber instead of float, then the change from one machine to the other is as simple as changing

ch2/typedef.C

```
// Small machine
typedef short int intNumber;
typedef float floatNumber;
```

to

ch2/typedef.C

```
// Large machine
typedef long int intNumber;
typedef double floatNumber;
```

and recompiling the program. Without the use of typedef declarations, you would have to locate every short int and change it to long int and every float and change it to double.

2.9 Arrays

An *array* is an ordered collection of objects, called *array elements*, all of the same type. Individual elements are selected by numeric subscripts. A FORTRAN array might be declared

ch2/simple.f

```
DIMENSION X(100)
```

with the first and last elements referred to as `X(1)` and `X(100)`, respectively. A similar array in C++ would be declared

ch2/simplecpp.C

```
float x[100];
```

and the first and last elements referred to as `x[0]` and `x[99]`.

- *Warning*: A C++ array of size `n` is indexed by the subscripts `0 . . . n-1`.

An array can be initialized by specifying an initializer list in its declaration, one value per element:

ch2/simplecpp.C

```
float x[3] = {1.1, 2.2, 3.3};
```

If the size of the array is omitted, it is taken from the number of items in the initializer list. For example,

ch2/simplecpp.C

```
float x[] = {1.1, 2.2, 3.3};
```

specifies a three-element array.

Multidimensional arrays are declared and indexed with multiple subscripts, as illustrated by the following code for multiplying 3 × 3 matrices:

ch2/mat3by3mult.C

```
float m[3][3], m1[3][3], m2[3][3];
double sum;     // Accumulate in double precision
// Code that initializes m1 and m2 ...

// m = m1 * m2
for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++) {
            sum = 0.0;
            for (int k = 0; k < 3; k++) {
                  sum += m1[i][k] * m2[k][j];
            }
            m[i][j] = sum;
      }
}
```

Each `for` loop declares an `int` variable—the C++ analog of a FORTRAN `DO` variable—to use for controlling the loop and indexing the matrix elements. Since C++ arrays are indexed starting at `0`, not `1` as in FORTRAN, each loop counts from `0` to 2, inclusive. The postincrement operator (`++`) conveniently increments the loop variables. Note that an element of a two-dimensional array is accessed with a pair of square bracketed numbers, as in `m[1][2]`.

	Declaration	*First*	*Last*	*Initializer*
FORTRAN Array	INTEGER X(3)	X(1)	X(3)	DATA X/7,8,9/
C++ Array	int x[3]	x[0]	x[2]	={7,8,9}
FORTRAN Matrix	INTEGER M(2,2)	M(1,1)	M(2,2)	DATA M/1,3,2,4/
C++ Matrix	int m[2][2]	m[0][0]	m[1][1]	={ {1,2}, {3,4} }

Table 2.10 Comparison of FORTRAN and C++ Array Declaration and Use. Analogous declarations, subscripts, and initializations are shown for both a 3-element array and a 2 × 2 matrix. FORTRAN arrays use 1-origin subscripting and the elements are stored column-wise; C++ arrays use 0-origin subscripting and the elements are stored row-wise.

- *Warning*: m[1, 2] is a valid C++ expression, but it does not access a multidimensional array.

Instead the expression 1, 2 is an example of the comma operator, which evaluates its operands in left-to-right order and has as its result the result of evaluating the right operand. (We won't have much use for the comma operator.)

As the declaration and subscripting notations suggest, multidimensional arrays are arrays of arrays. This is apparent in their initialization syntax:

ch2/simplecpp.C

```
int m[2][3] = {
                {1, 2, 3},
                {4, 5, 6}
              };
```

In this example, m consists of two elements (rows), each a one-dimensional array of three elements. Notice that the values appear row-wise, not column-wise as they would in a FORTRAN DATA statement. This arrangement reflects the way C++ arrays are stored in memory—using row-major order instead of column-major as in FORTRAN.

Table 2.10 summarizes the declaration and use of C++ and FORTRAN arrays and two-dimensional matrices. One-dimensional arrays with size determined

at runtime are discussed in Section 2.13, and a flexible system of programmer-defined arrays is introduced in Chapter 13.

2.10 Pointers

A *pointer* is an object that refers to another object. The actual bits in a pointer object depend on the computer running the C++ program; typically the bits represent a machine memory address. Pointers are the basic element needed to manipulate complex data structures. This section introduces the basic concepts of pointers; Section 2.11 describes the relationship between pointers and arrays; Section 2.13 describes the use of pointers to refer to objects whose size is known only at runtime.

A pointer type is formed from any other type by adding an asterisk to the type. For example,

ch2/ptrs.C

```
int* p;
```

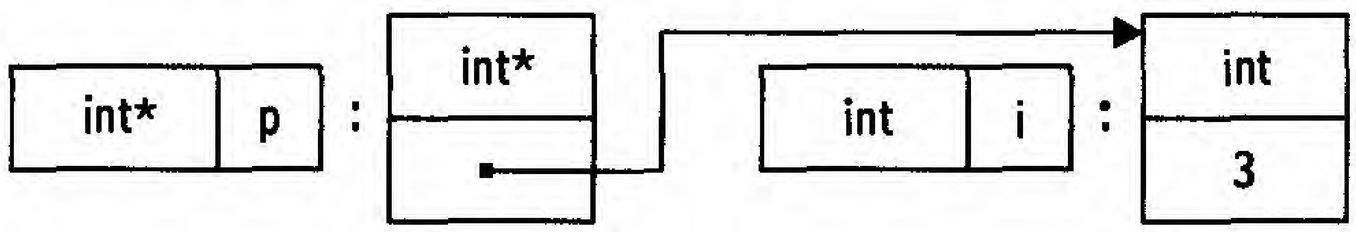
declares the variable p to be of type pointer to int or int pointer. (The asterisk need not be adjacent to the type; see Notes and Comments 2.3.) This means that the object for p points to an object of type int. The particular object it points to is determined by the value of p, which can be set by assignment or initialization.

The *address-of operator*, & , yields a pointer to its operand. Thus

ch2/ptrs.C

```
int i = 3;
p = &i;
```

causes p to point at the object associated with i. Schematically

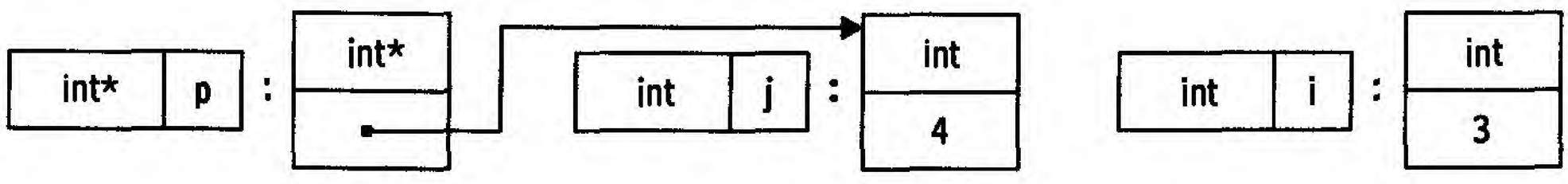


After executing

ch2/ptrs.C

```
int j = 4;
p = &j;
```

p points to the object associated with j:



Given a pointer to an object, the *indirection* operator * accesses the object. Assuming p has been set as just described,

ch2/ptrs.C
```
cout << *p << endl;
```

writes 4, and

ch2/ptrs.C
```
*p = 5;
cout << *p << " " << i << " " << j << endl;
```

writes 5 3 5. We read *p as "the object pointed to by p." The crucial concept is that *p points to exactly the same object with which j is associated. The assignment *p = 5 alters that object just as if we had written j = 5.

A pointer may point to different objects at different times; in some cases, we want a pointer to point to nothing. The *null pointer* is guaranteed to be distinguishable from the pointer to any valid object or function and thus acts as a pointer to nothing. The literal constant 0, used as a pointer, is a null pointer. Continuing the preceding example,

ch2/ptrs.C
```
if (p != 0) {
    cout << "Pointer p points at " << *p << endl;
}
```

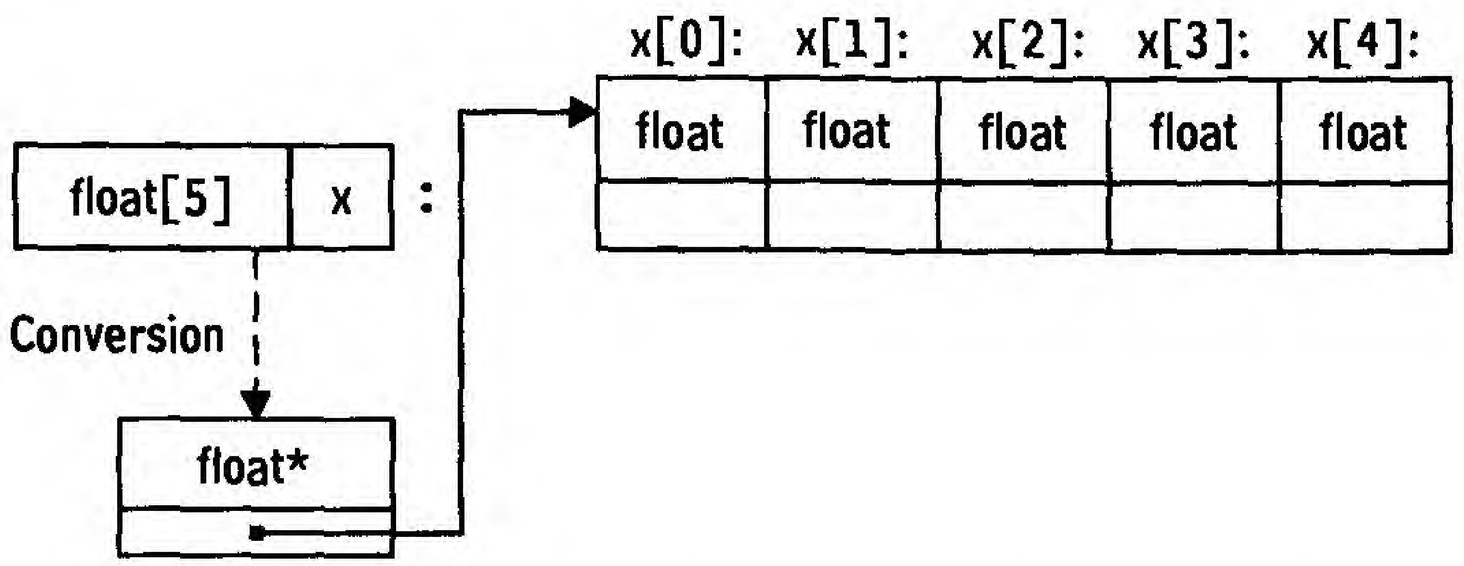
writes Pointer p points at 5. It is an error to use the indirection operator on a null pointer.

2.11 Pointers and Arrays

Pointers and arrays are intimately related in C++. The declaration

ch2/ptrArray.C
```
float x[5];
```

associates with the variable x an array of five numbers, and x has the type array of float. However, whenever the variable x is used in an expression—except as the operand of the address-of operator & , the sizeof operator, or to initialize a reference (Section 2.15)—it is converted into a pointer to the first element of the array:



This picture is conceptual: x is still of type array of float and there is no pointer object. In other words, x is used as if it were of type float* (pointer to float) and

x[0]—the 0^{th} element of array x—is the same as *x—the object pointed to by the variable x. An array's elements can be accessed either through a subscript or by indirection through a pointer.

C++ defines arithmetic on pointers. An integer can be added to (or subtracted from) a pointer to an array element. The integer acts as an offset to the pointer, and the sum (or difference) is a pointer of the same type, which points to the element of the same array the specified number of elements away. For example, x+1 is a pointer to the element that is one element past the element pointed to by x; x points to x[0] and x+1 points to x[1]. In general, x[i] and *(x+i) are equivalent, as are & x[i] and x+i.

Two pointers that point to elements of the same array can be compared using the relational operators <, <=, >=, and >. The result is defined by comparison of the equivalent subscripts. For example, the following code reverses the elements of an array:

ch2/array-reverse.C

```
float x[10];
// ... initialize x ...
float* left = &x[0];
float* right = &x[9];
while (left < right) {
    float temp = *left;
    *left++ = *right;
    *right-- = temp;
}
```

The pointers left and right point to the leftmost and rightmost elements that have not yet been swapped. Each iteration of the loop swaps those two elements and increments the left pointer (moves it to the right) and decrements the right pointer (moves it to the left). The loop terminates when the left pointer is no longer to the left of the right pointer. Combining the indirection operator with postincrement or postdecrement operators, as in *left++ and *right--, is common in C and C++ programming. The postincrement and postdecrement operators have higher precedence than the indirection operator; see Table 2.7. Hence they apply to the pointers, not to the objects given by the indirection operator. At first these compact expressions require extra attention; with practice they become second nature.

The equality operators == and != can also be used to compare any two pointers of the same type, as in this code to set the elements of a 10-element array to 0:

ch2/array-zero.C

```
float* p = &x[10];
while (p != x) *--p = 0.0;
```

Here the pointer is decremented before use, not after, as in the previous example. The expression `&x[10]` yields the address immediately beyond the end of the last element of the array `x`. C++ guarantees that a pointer to an element of an array (e.g., `p`) can be compared to such an address, even though it would not be correct to write `*p` before decrementing `p`.

2.12 const Pointers and Pointers to const Objects

Pointers provide a route to an object. To ensure that a `const` object remains constant when accessed through a pointer, C++ only allows pointers-to-const types to point to const objects:

ch2/ptrconst.C

```
const float pi = 3.14159;    // Constant object
const float* p = &pi;
```

This declares `p` to be of type pointer to `const float`, meaning that it is a pointer to a `float` that is constant. With this declaration, `*p` is a `const float` that cannot be modified. For example, the following code gives a compile error:

ch2/ptrconst.C

```
*p = 2.0;                    // WRONG
```

A pointer-to-const type object may point to a non-const object. Such an object can be modified, but not through the pointer.

For the same reasons that we protect ourselves against erroneous modifications of constant data with `const` declarations, we might want to protect against erroneous modifications of pointers by making them constant:

ch2/ptrconst.C

```
float a;
float* const p1 = &a;
```

This declares `p1` to be of type constant pointer to `float`, meaning that the pointer itself is constant, but the object is not. Thus

ch2/ptrconst.C

```
*p1 = 10.0;
```

is correct, but

ch2/ptrconst.C

```
float b;
p1 = &b;                     // WRONG
```

is not because it tries to change the value of `p1`. Finally, we might want both the pointer and the object to remain constant:

ch2/ptrconst.C

```
const float e = 2.718281828;
const float* const p2 = &e;
```

In this case, neither `e` nor `p2` can be modified.

The syntax for declaring constant pointers and pointers to constant objects is arcane, but here is a rule that might help:

- Read complex declarations from the identifier back toward the beginning of the statement.

In `const float* p`, read "`p` is a pointer to a `float` that is const"; in `float* const cp`, read "`cp` is a const pointer to a `float`"; in `const float* const p2`, read "`p2` is a const pointer to a `float` that is const."

2.13 Runtime Array Size

Through the use of pointers, C++ allows the number of elements in an array to be determined when the array is created. The code

ch2/ptrArray.C

```
float* const x = new float[n];
```

creates an array of `n` floats and sets `x` to point at the array's first element, as follows:

1. The declaration `float* const x` declares `x` to be a constant pointer to a `float`.
2. The `new` operator creates an array of `n` objects of type `float` and returns a pointer to the first element of the array.
3. `x` is initialized to the pointer to the newly created array of `n` floats.

It is important to understand that the preceding C++ code is different from the FORTRAN code:

ch2/simple.f

```
      SUBROUTINE F(X, N)
      DIMENSION X(N)
*     ...
      END
```

The maximum size of the array `X` is determined by the caller of `F` (or one of its callers), which must somewhere have a `DIMENSION` statement with a specific array size in it; the C++ code need not have any such fixed size, and `n` could be computed or read from input.

Let's see how to use this capability by writing a function that fits a straight line to a set of data points (x-y pairs) of arbitrary size. To keep matters simple, we shall ignore the statistical issues that must be considered when analyzing data; see, for example, [93, Section 15.2]. The function will find coefficients a and b so that $y = a + bx$ is the best-fitting straight-line model for the data points (under simplifying assumptions). The input will consist of a count of the number of data points, followed by that many x-y pairs. The coefficients are given by

$$a = \frac{S_y - S_x b}{n}$$

$$b = \frac{1}{S_{tt}} \sum_{i=1}^{n} t_i y_i \,,$$

where $S_x = \sum_{i=1}^{n} x_i$, $S_y = \sum_{i=1}^{n} y_i$, $t_i = x_i - S_x/n$, and $S_{tt} = \sum_{i=1}^{n} t_i^2$.

The function begins by reading from the standard input the number of data points and then creating two arrays of the appropriate size:

ch2/linefit.C

```
#include <iostream.h>

void linefit() {
    // Create arrays with the desired number of elements
    int n;
    cin >> n;
    float* const x = new float[n];
    float* const y = new float[n];
```

Once the two arrays have been created, they can be used as ordinary arrays. In particular, the normal array subscript operations can be used (and, for clarity, should be used) because of the relationship between pointers and arrays. Reading the data points and computing the sums is straightforward:

ch2/linefit.C

```
// Read the data points
for (int i = 0; i < n; i++) {
    cin >> x[i] >> y[i];
}

// Accumulate sums Sx and Sy in double precision
double sx = 0.0;
double sy = 0.0;
for (i = 0; i < n; i++) {
    sx += x[i];
    sy += y[i];
}

// Compute coefficients
double sx_over_n = sx / n;
double stt = 0.0;
double b = 0.0;
```

```
for (i = 0; i < n; i++) {
    double ti = x[i] - sx_over_n;
    stt += ti * ti;
    b += ti * y[i];
}
b /= stt;
double a = (sy - sx * b) / n;
```

Once the arrays are no longer needed, they can be discarded using the delete[ ] operator:

ch2/linefit.C

```
delete [ ] x;
delete [ ] y;
```

The empty square brackets ([ ]) signify that an array is being deleted. The function completes by writing the coefficients

ch2/linefit.C

```
    cout << a << " " << b << endl;
}
```

Table 2.11 summarizes the operators that manipulate pointers and arrays.

Operator	*Purpose*
*p	Indirection
&x	Pointer to object
a[i]	Array subscript
p->m	Class member selection
++p	Preincrement to next element
p++	Postincrement to next element
--p	Predecrement to previous element
p--	Postdecrement to previous element
p+=n	Increment by n elements
p-=n	Decrement by n elements
p+n	Offset by n elements
p-n	Offset by n elements
new T	Allocate object
new T[n]	Allocate array of n objects
delete p	Delete object
delete [ ] p	Delete array of objects

Table 2.11 Pointer Operators

2.14 Character Strings

Character strings are a special case of arrays and array initialization. For example, the array

ch2/charArray.C

```
char hello1[] = {'H', 'i'};
```

declares an array containing two char elements, initialized to the values 'H' and 'i'. An array of characters can also be specified between double quotes:

ch2/charArray.C

```
char hello2[] = "Hi";
```

This second array actually contains three elements, not two, because C++ adds an additional array element initialized to the *null character* ('\0'). By convention, the null character marks the end of a character string. In

ch2/charArray.C

```
char name[15] = "Isaac Newton";
```

the array name is initialized to

I	s	a	a	c		N	e	w	t	o	n	\0	?	?

where the question marks indicate uninitialized array elements. The code

ch2/charArray.C

```
name[12] = '\'';
name[13] = 's';
name[14] = '\0';
```

changes the character string to "Isaac Newton's". The ANSI standard C library provides subroutines for more conveniently manipulating character strings that adhere to the null character termination convention. See Section 18.2 for a more sophisticated mechanism for handling character strings.

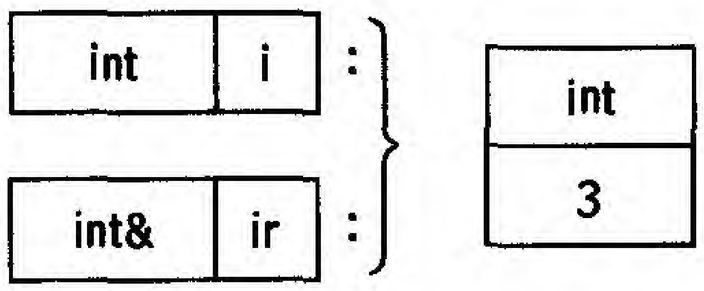
2.15 References

A *reference* is an alias for an object of a specified type. For example, the declarations

ch2/refs.C

```
int i = 3;          // Variable
int& ir = i;        // Reference to variable i
```

declare a variable, i, and a reference to it, ir. The second declaration declares ir to be of type int& , which means "reference to int." A reference introduces a new name for an existing object, so that after the preceding code we have



The connection between a reference and an object is formed when the reference is initialized; hence references must always be initialized with some object, and they refer thereafter to that object.

Given the preceding declarations of i and ir, executing

ch2/refs.C

```
cout << ir << endl;
ir = 4;
```

prints 3 and then changes the value of i to 4. Now executing

ch2/refs.C

```
cout << ir << endl;
```

prints 4. Notice that assigning to a reference alters the object referred to, not the reference itself.

Both pointers and references provide access to an object, but they are different in important ways. A pointer points to an object, the pointer can be changed to point to a different object, the indirection operator must be used to obtain the object, and storage is used for the pointer's value. A reference acts like another name for the object, it can point only at one object, no operator is needed for access, and no storage need be allocated.

The general form of a reference declaration is a type followed by an ampersand (&), a name, and an initialization. If the type is, for example, float, then the reference type will be float&. Thus float& fr declares the name fr to be a variable of type float& read as reference to float or float reference. As with pointers, there are also references to const objects. For example, in

ch2/refs.C

```
const float pi = 3.14159;
const float& unit_circle_area = pi;
```

unit_circle_area is a const reference to the constant object pi. It is also possible to have a const reference to a non-const object:

ch2/refs.C

```
float x;
const float& xr = x;
x = 5.0;
```

The compiler will flag any attempt to alter the value of x by assigning to xr, but it will allow assignment to x itself.

2.16 Functions

C++ functions are analogous to FORTRAN SUBROUTINE and FUNCTION subprograms. They contain statements expressing part of an algorithm, either as an aid to program organization or to reuse the code.

2.16.1 Syntax

A C++ function definition has the following form:

```
return-value-type function-name(argument-list) {
    function-body
}
```

For example,

ch2/coulombsLaw-onefile.C

```
double coulombsLaw(double q1, double q2, double r) {
    // Coulomb's law for the force acting on two point charges
    // q1 and q2 at a distance r. MKS units are used.
    const double k = 8.9875e9; // nt-m**2/coul**2
    return k * q1 * q2 / (r * r);
}

int main() {
    cout << coulombsLaw(1.6e-19, 1.6e-19, 5.3e-11) << " newtons" << endl;
    return 0;
}
```

defines a function named coulombsLaw; it returns a double (the *return-value-type*) and expects three arguments, each with type double. The *function-body* has two comments, declares and initializes a double, and computes Coulomb's law in the expression of the return statement. Like the FORTRAN RETURN statement, the C++ return statement terminates execution of the function and returns control to the caller. The C++ return also specifies the value to be returned, here a double or something that can be converted to a double. (In FORTRAN, the value is given by assigning to the function's name.)

A C++ function is called when its name appears followed by matching parentheses, possibly containing a list of arguments. In the preceding example, the main() function (cf. Section 2.1) prints the electrical attractive force between a hydrogen atom's electron and proton, by calling the coulombsLaw() function with a list of three arguments.

C++ functions need not return a value, in which case the keyword void is used as the *return-value-type* and the return statement is either omitted or used without

specifying a value. Likewise, a function need not have arguments, in which case the *argument-list* is omitted, leaving an empty pair of parentheses following the *function-name*. (Alternatively, the *argument-list* can consist of the single keyword void; see Section 3.10.)

2.16.2 Function Prototypes

Most programs are too large to keep in one source file that is recompiled in its entirety each time the program is changed. Thus it is common for a function defined in one file to call a function defined in another file. FORTRAN programs also have this characteristic, but there is an important difference: C++ checks the types of function arguments and checks for type-compatible operations on the objects returned from functions. To do this, C++ requires each function to be declared before it can be called. That is, you must tell the compiler the name of the function, the type of its arguments, and the return type before you can call the function. In the preceding example, the function definition, which appeared before the call in main(), did this. But if we move the function definition to its own file, we must insert a function declaration, in the form of a *function prototype*, before the function call.

The function prototype begins with the keyword extern, signifying that we are declaring but not defining (giving the code for) a function, followed by the same return type, function name, and argument declaration sequence we used in the function definition. The prototype ends with a semicolon, where the function body opening brace would be:

ch2/coulombsLaw.h

```
extern double coulombsLaw(double q1, double q2, double r);
```

To ensure consistency among the function definition, the function prototype, and the function call, function prototypes are usually put into header files. The header file is then included (via #include) in both the file that calls the function and the file that defines the function. In our example, the file that calls coulombsLaw() looks like this:

ch2/coulombUse.C

```
#include <iostream.h>
#include "ch2/coulombsLaw.h"

int main() {
    cout << coulombsLaw(1.6e-19, 1.6e-19, 5.3e-11) << " newtons" << endl;
    return 0;
}
```

The file that defines the function looks like this:

ch2/coulombsLaw.C

```
#include "ch2/coulombsLaw.h"

double coulombsLaw(double q1, double q2, double r) {
    // Coulomb's law for the force acting on two point charges
    // q1 and q2 at a distance r. MKS units are used.
    const float k = 8.9875e9;  // nt-m**2/coul**2
    return k * q1 * q2 / (r * r);
}
```

When coulombsLaw.C is compiled, the function prototype and the function definition will be checked for consistency; likewise the function prototype and the function call will be checked when coulombUse.C is compiled. In this way, the compiler ensures that the arguments to functions have the correct type and number.

- Put function prototypes in header files. Include the header files in the file that defines a function and in every file containing calls to that function.

As this example illustrates, there are two forms of the #include directive. A system-provided header file, like iostream.h, should be included using the angle bracket form:

ch2/includes.C

```
#include <iostream.h>
```

Include your own header files with the other form:

ch2/includes.C

```
#include "ch2/coulombsLaw.h"
```

Each compiler can define the details of what the two different forms mean, but it is intended that the angle bracket form look for the file in standard places first. The version with double quotes usually means to start looking in the directory of the including file first.

The compiler also uses the information in function prototypes to convert the arguments supplied in a function call to match those in the function prototype. For example,

ch2/coulombUse.C

```
cout << coulombsLaw(1.6e-19, 1.6e-19, 1) << " newtons" << endl;
```

is correct even though the third argument has the wrong type. The compiler knows that the third argument should be a double, so it converts the int 1 to the double 1.0 before calling coulombsLaw. Automatic type conversion (cf. the usual arithmetic conversions described on page 23) and type checking by the C++ compiler eliminate two common sources of programming errors.

2.16.3 Inline Functions

For most functions, the time required to execute the function's body dominates the total time taken by a call to the function. In other words, the overhead of calling the function is insignificant compared to the computation done by the function. Occasionally, however, the overhead is a significant fraction of the total function call cost. If such a function is called many times, the overhead can be a significant fraction of a program's total execution time.

When function call overhead is significant, a function definition can be preceded by the keyword inline. This instructs the compiler to attempt to generate inline code for calls to the function. Here is an example:

ch2/doubleSqr.C

```
inline double sqr(double x) {
    return x * x;
}
```

When this function is called, we expect the compiler to generate inline code without function call overhead. The inline keyword is a suggestion to the compiler; a C++ compiler is not required to obey the suggestion. Any decent compiler will generate inline code for a function like sqr(). There is, however, considerable variation among compilers in how they handle more complex inline functions, especially those containing loops.

The definition of an inline function must have been seen by the compiler before the function is called. It is not enough for the compiler to have seen a function prototype; without the definition itself, the compiler cannot generate inline code. Thus an inline function definition is usually put in a header file, in place of the function prototype.

Although inline functions eliminate function call overhead, they can introduce other overheads. When a function is inlined, its code is duplicated for each call. Excessive use of inline may result in large programs. Large programs may have poor instruction cache behavior or cause excessive paging in virtual memory systems.

Overusing inline functions can also lengthen compile and link times: When the definition of an inline function is changed, every function that calls it must be recompiled; when the body of a noninline function is changed, only that function need be recompiled. Moreover, the definitions of inline functions may require other header files, and changes in those files may cause recompilation of the caller of an inline function. This leads us to recommend the following:

- Use inline functions sparingly.

On the other hand, not inlining small functions that do little computation can make programs both bigger and slower. To complicate matters, the decision is somewhat machine and compiler dependent.

One approach is not to inline any functions until the program runs and its performance can be measured. Functions that are performance bottlenecks can then be inlined. But one must understand what is being measured. Take the extreme in which a function f() repeatedly calls a function g() with an empty body: g() will not appear to take any time, yet making it inline could greatly improve the running time of f(). In deciding which functions to inline, it is important to consider both the time taken by the candidate function and the time taken by functions that call it.

Another factor is ease of debugging. On some systems, debugging an inline function is difficult because the function doesn't exist at runtime. On such systems, it might be appropriate not to inline any functions until the entire program is running.

We subscribe to a modified measure-before-inlining approach. We generally inline functions that just return a built-in type or a reference. We also generally inline functions that just return the result of calling another function. We sometimes inline functions that do a simple computation and are likely to be called from within loops in other functions. Beyond this, we measure first.

2.16.4 Variable Scope, Initialization, and Lifetime

A variable can be declared and initialized anywhere in a block (function bodies and the statements grouped in braces for if statements and loops are blocks). A variable declared in a block is called a *local variable,* and its name is said to be in the *local scope*. The name is known from the point of declaration to the end of the block. A local variable hides variables of the same name in containing scopes:

ch2/scope.C

```
void f() {
    float temp = 1.1;
    int a;
    int b;
    cin >> a >> b;

    if (a < b) {
        int temp = a;    // This "temp" hides the one in function scope.
        cout << 2 * temp << endl;
    }    // Block ends; local "temp" deleted.
    else {
        int temp = b;    // Another "temp" hides the one in function scope.
        cout << 3 * temp << endl;
    }

    cout << a * b + temp << endl;
}
```

The three variables named temp in this code are all different. Within the blocks associated with the if and else statements, uses of the name temp refer to the local variable temp declared in that block; the use of temp outside the if and else blocks refers to the local variable declared at the beginning of the function.

The object associated with each such local variable is created and initialized when the declaration is encountered in the program execution sequence. The association is broken at the end of the block and the object is discarded. Thus each call to a function causes new versions of its local variables to be created, initialized, used, and discarded. The same is true of local variables in other (nonfunction) blocks:

ch2/mat3by3mult2.C

```
float m[3][3], m1[3][3], m2[3][3];
// Code that initializes m1 and m2 ...

// m = m1 * m2
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        double sum = 0.0;
        for (int k = 0; k < 3; k++) {
            sum += m1[i][k] * m2[k][j];
        }
        m[i][j] = sum;
    }
}
```

Here we have rewritten the matrix multiply code from page 41 using a local variable sum in a for loop's block. The object associated with sum is created and initialized each time the block is begun. Proximity of a variable's type declaration, initialization, and use aid program readabililty:

- Declare local variables near their first use.

2.16.5 Arguments

The arguments that appear in the function header are called *formal arguments*, and the arguments passed to the function are called *actual arguments*. Each formal argument is a local variable of the function, and each formal argument is initialized with its corresponding actual argument. For example,

ch2/funcarg.C

```
void f(int i, float x, float* a) {
    i = 100;
    x = 101.0;
    a[0] = 0.0;
}
```

declares three formal arguments: i, x, and a. Calling the function like this

ch2/funcarg.C

```
int j = 1;
int k = 2;
float y[ ] = {3.0, 4.0, 5.0};
// ...
f(j, k, y);
```

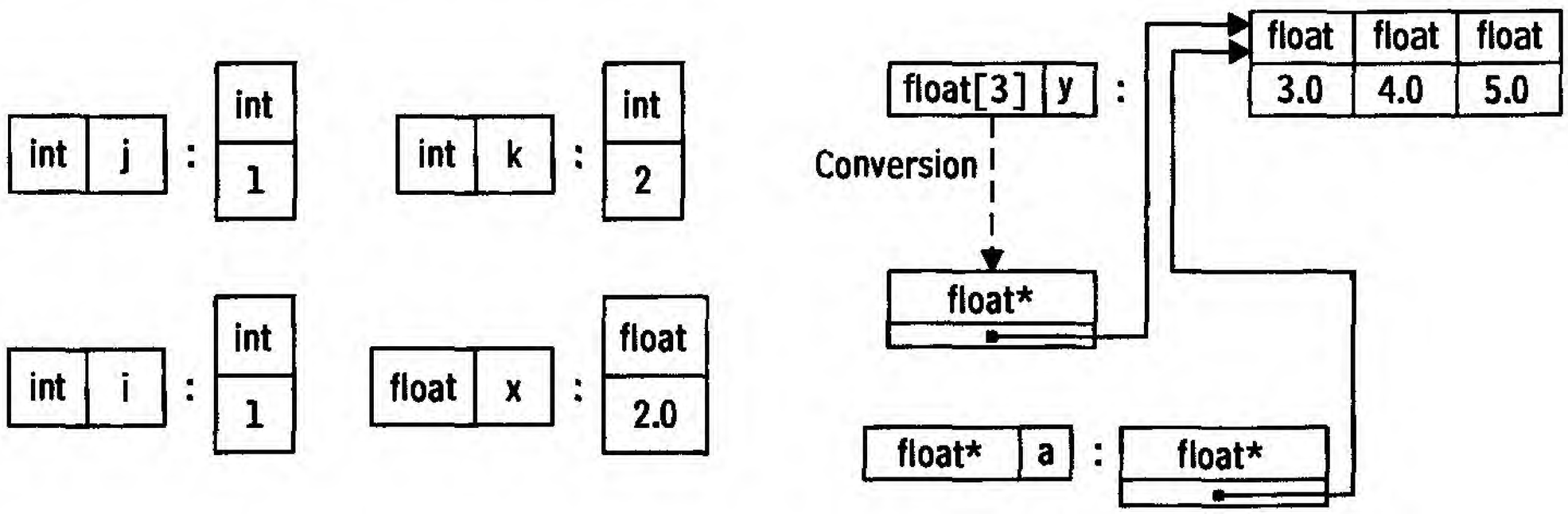
causes actions equivalent to

ch2/funcarg.C

```
int i = j;
float x = k;
float* a = y;

i = 100;
x = 101.0;
a[0] = 0.0;
```

First, the three formal arguments are initialized from their corresponding actual arguments. To initialize x, k is converted from an int to a float; to initialize a, y is converted from type array of float to a float* that points to the first element of y (cf. Section 2.11). After initialization of the formal arguments, the association between variables and objects looks like this:



In this example, the assignments to i and x alter those variables within the f() function but have no effect on the values of the actual arguments j and k in the calling function; the arguments are said to be *passed by value*. The assignment to i and to x has no effect in this function.

As we saw in Section 2.2, FORTRAN is different: It creates a (temporary) association between a formal argument and the object associated with the actual argument. FORTRAN's behavior, in which changes to formal arguments affect the

corresponding actual arguments, is called *call by reference*; the arguments are said to be *passed by reference*.

Passing an array as an actual argument is tantamount to passing a pointer to the array's first element. Indeed an array formal argument can be declared either as a pointer, like the third argument of the function f() on page 57, or as an array, like float a[ ]. The pointer is passed by value, but the value of the pointer points to data in the calling function. The assignment to a[0] alters y[0] because a points to the first element of y. In other words, call by value of a pointer to the first element of an array passes a copy of the pointer's value, not a copy of the array's elements. Since the copy of the pointer points to the array elements, they can be changed through the copy of the pointer. Thus call by value of the pointer to the first element of an array results in call by reference of the array's elements.

To alter an argument's value, or to avoid copying larger nonarray objects (the class objects introduced in Chapter 4), reference formal arguments can be used:

ch2/funcarg.C

```
void swap(int& i1, int& i2) {
    // Swap arguments
    int temp = i1;
    i1 = i2;
    i2 = temp;
}
```

The formal arguments i1 and i2 are references (cf. Section 2.15). Initializing them with the actual arguments, like this

ch2/funcarg.C

```
int c = 3;
int d = 4;
swap(c, d);
// c == 4 and d == 3
```

forms an association between the formal argument names and the corresponding actual argument objects in the calling function. Then changes to the formal arguments affect the actual arguments and we have call by reference, as in FORTRAN.

A more detailed discussion of function arguments appears in Section 5.3.

2.16.6 Recursion

The independence of local variables across different calls of a function makes it possible for a function to call itself, a process called *recursion*. As an example, consider computing the Stirling number of the second kind $\left\{ {n \atop k} \right\}$, defined by [52, Section 6.1] as follows:

$$\left\{ {n \atop k} \right\} = \begin{cases} 0 & n < k, \\ 0 & k = 0, n > 0, \\ 1 & n = k, \\ k\left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\} & n > 0. \end{cases}$$

The value of $\left\{ {n \atop k} \right\}$ is the number of ways to partition a set of n things into k nonempty subsets. With recursion, we can implement the definition directly:

ch2/Stirling.C

```
int stirling(int n, int k) {
    if (n < k) return 0;
    if (k == 0 && n > 0) return 0;
    if (n == k) return 1;
    return k * stirling(n-1, k) + stirling(n-1, k-1);
}
```

Many mathematical functions are implemented easily using recursion. However, recursive functions can be less efficient than their nonrecursive equivalents, and some recurrence relations are numerically unstable (see [93, Section 5.5]). Recursion is also a powerful tool for manipulating data structures.

2.16.7 The main() Function

Every C++ program contains one special function, main(), the starting point for program execution. The simplest function prototype for this function is int main(), meaning that it is a function without arguments returning an integer. The return value, set as usual by the return statement, can be tested in most operating systems as the program's return code. For portability, use the constants EXIT_SUCCESS and EXIT_FAILURE, defined in the standard header file stdlib.h, for return codes.

The main() function can also have other forms, in which its arguments give the number of arguments (words) on the program execution command line (argument count), the arguments themselves as entries in an array of pointers to character strings (argument values), and, optionally, other information. Several standard methods of processing the command line words exist, but the specifics depend on operating system conventions. Most Unix systems provide the getopt() function for processing Unix-style command line arguments that consist of a sequence of flags and words. For example, here is a program that prints a list of one-character command line flags, each in the range 'a' to 'g':

ch2/commandLine.C

```
#include <iostream.h>
#include <stdlib.h>
```

```
int main(int argc, char* argv[ ]) {
    cout << "Flags found: ";
    int flag;
    while ((flag = getopt(argc, argv, "abcdefg")) != EOF) {
        char flagc = flag;
        cout << flagc << " ";
    }
    cout << endl;
    return EXIT_SUCCESS;
}
```

Other standard functions are available on other systems. For consistency,

- Use standard functions for processing command line arguments.

2.16.8 Summary

We have discussed the elementary aspects of C++ functions. We revisit much of this material in greater detail in Chapter 5, where we also introduce new aspects of C++ functions.

All of the functions we have described thus far are *global functions*. They manipulate data, but they are not attached to a data structure. C++ programs are not organized around such functions, but around *classes*, an idea we turn to in Chapter 4. Chapter 3 is intended for experienced C programmers. Unless you are interested in understanding differences between C++ and C, you should proceed directly to Chapter 4.

2.17 Notes and Comments

2.1 Consistent indentation aids code readability. We have chosen one style, but there are other possibilities, each with its own adherents. One popular style aligns braces vertically:

ch2/vertstyle.C

```
if (a < b || b > 10)
    {
    a = 3;
    b = 4;
    }
```

This style shows the levels of nesting more clearly at the expense of fitting less code and thus less context in a given space. This was the deciding factor in choosing a code style for this book. A thoughtful discussion of indentation styles appears in [104, Chapter 6].

2.2 If you think that referring to the first element of an array by the index 1 is somehow fundamentally superior to referring to it by the index 0, pick up any mathematics book containing series and see how many of them are written with an initial index of 0. The choice of initial index is a convention; one convention will be more convenient in some applications than others. C++ allows programmer-defined arrays to be created with arbitrary first indexes. See Exercise 4.12.

2.3 The formal syntax of pointer declarators associates the * with the name of the variable being declared. In fact, C programmers often write

ch2/ptrdclstyles.C

```
int *p;
```

instead of the

ch2/ptrdclstyles.C

```
int* p;
```

that we write. Our style, also used in some other C++ books, seems more natural: p is an int*, a pointer to an int. The C style mirrors the use of p in an expression: The expression *p has type int. The C style is preferable if you declare several variables in one declaration statement, like this:

ch2/ptrdclstyles.C

```
int* p1, p2;
```

Here p1 is a pointer to an int, but p2 is an int, not a pointer to an int. Declaring one variable per statement avoids this pitfall.

2.4 Our description in Section 2.8 of the syntax of attributes and declarators is a simplification of the actual rules. C++'s declarator syntax allows complicated types to be declared; however, "the declarator syntax of C – and therefore of C++ – is widely recognized as being unnecessarily hard to read and write" [44, page 132]. Fortunately, it is rarely necessary to write declarators for complicated types, and typedef statements can be used to simplify the syntax of those that must be written. A particularly lucid description of the syntax of C declarators, which are the basis for C++ declarators, appears in [58].

2.5 This chapter describes the low-level array and pointer facilities that are built into C++. Programs can, and in most cases should, be written with programmer-defined array and pointer types. See Sections 4.2 and 4.3 and Chapter 13.

2.6 The importance of pointers and runtime array size has led to the inclusion of both in FORTRAN-90 [78].

2.7 The bytes in a stream can be interpreted as containing either binary data or character data. Binary data correspond directly to the internal representation of data in the computer and are inherently nonportable; FORTRAN's unformatted READ and WRITE statements read and write binary data. Character data, on the other hand, are converted from internal representation to a human readable and printable form. We discussed character data in Section 2.4; iostream also supports binary data.

The functions for manipulating binary data are portable across implementations of the iostream classes; the exact form of the data depends on the hardware. Moreover,

since some FORTRAN compilers insert control information into the binary data written by unformatted WRITE statements, C++ code that writes FORTRAN-compatible binary data is inherently machine and FORTRAN compiler dependent.

2.18 Exercises

2.1 Type in intercepts.C (page 16), then compile and run it.

2.2 Write intercepts.f by translating each line of intercepts.C (page 16) into FORTRAN. Since intercepts.C simply implements a formula, formula translation should be perfect for this program. What percentage of the lines in intercepts.f are not formula translation?

2.3 Using the diagram style of Section 2.2, sketch the variables, objects, and types in the following program:

ch2/ex-varsketch.C

```
void s(float& a, float b) {
    a = b;
}

int main() {
    float x = 12.1;
    s(x,4.2);
    return 0;
}
```

2.4 The character comparison code on page 25 won't compile if the parentheses are removed. Explain why.

2.5 Is it legal for the following statements to appear together? If not, why?

ch2/enum.C

```
enum Color { red, orange, yellow, green, blue, indigo, violet };
enum Crt_colors { red, green, blue };
```

2.6 What does this program print?

ch2/ex-while.C

```
#include <iostream.h>
int main() {
    int i = 13;
    while (i) {
        i /= 2;
        cout << i << endl;
    }
    return 0;
}
```

2.7 Given

ch2/ex-ptrs.C

```
float x[5];
```

what do each of the following lines mean? Are they syntactically correct?

ch2/[redacted]trs.C

```
x[0] = 1.0;
x[4] = 5.0;
*(x+4) = 6.0;
*(&x[3]+1) = 7.0;
*x+4 = 8.0;
(&x)[4] = 9.0;
x[5] = 2.0;
x++; x[4] = 3.0;
int i = x[3] < x[4];
int j = x+3 < x+4;
```

2.8 What do the following statements mean?

ch2/ex-refs.C

```
int i = 5;
int& ir = i;
int* ip = &ir;
int*& ipr = ip;
const int& cir = i;
```

2.9 Rewrite the array reversal example on page 45 using array subscripting instead of pointer manipulation. Compare on your computer the running time of the array subscripting and pointer versions using an array with 10,000 elements. Compare the readability of the two versions.

2.10 Write a function that computes the dot product of two n-element vectors v1 and v2. The function should have the following prototype:

ch2/dotproto.C

```
float dot(float v1[ ], float v2[ ], int n);
```

2.11 Using your solution to Exercise 2.10, write a program that reads, computes, and writes to cout an arbitrary number of dot products of vectors read from cin. Each pair of vectors is given in the input as a positive size n followed by the n elements of one vector, then the n elements of the second vector. The input does not contain a count of the number of dot products to be computed.

2.12 A polynomial

$$P(x) = \sum_{i=0}^{n-1} a_i x^i$$

should be evaluated using Horner's method:

$$P(x) = (\cdots((a_{n-1}x + a_{n-2})x + a_{n-3})x + \cdots)x + a_0.$$

Write a program that reads from cin the number of coefficients, n, followed by n coefficients, followed by an arbitrary number of x values. For each x value, compute and print $P(x)$.

2.13 Implement a nonrecursive function for computing Stirling numbers of the second kind (cf. Section 2.16.6). Hint: Use Stirling's triangle for subsets. Is this more efficient than the recursive function? Why or why not? Discuss the practical significance of any efficiency difference.

2.14 What would the function f() on page 56 print if given the input 3 4? What would it print if given 4 3?

2.15 What are the values of j, k, and y after executing the call to function f() on page 58?

CHAPTER 3

Basics for C Programmers

As INDICATED by its name, C++ extends C. This chapter describes the important differences between the part of C++ that is essentially C and C itself. We assume that you know C, so we focus on what C++ adds to it. If you don't know C, you should read Chapter 2 instead of this chapter, even if you don't know FORTRAN.

There are actually two programming languages called C: traditional C and ANSI C. The original definition of C, often called K&R C, appeared in the book *The C Programming Language* [65] by Brian Kernighan and Dennis Ritchie. After this, the language evolved gradually in minor ways into traditional C. In 1982, ANSI formed a committee to standardize the C programming language. The result of their effort, issued in 1989 as an ANSI standard [4American National Standards Institute], is ANSI C. ANSI C cleans up some aspects of traditional C and adds several new features, the most relevant to us being function prototypes.

Compilers for ANSI C have been available for more than five years. Therefore when we refer to the C programming language, we mean ANSI C.

3.1 A First Program

We start with a simple C++ program only slightly more complicated than Kernighan and Ritchie's [65] famous "hello, world" program. Our program reads and prints three floating point numbers:

ch3/regurgitate.C

```
#include <iostream.h>

int main() {
    // Read and print three floating point numbers
    float a, b, c;
    cin >> a >> b >> c;
    cout << a << ", " << b << ", " << c << endl;

    return 0;
}
```

This example illustrates a convention we use throughout the book: Each code fragment is annotated with the name of the file that contains it. (You can obtain these files; see Section 1.6.) When we want to intersperse explanatory prose with file contents, we simply continue the file later. You can find all code fragments contained in a file by looking for the file name in the Source File Index.

The first line of the example includes the header file iostream.h, which contains the declarations needed to use C++'s input and output operations. The iostream.h header file is analogous to the stdio.h header file one includes in C programs.

- Include iostream.h in any file that contains C++ input or output operations.

The first line in the body of main() is a comment. A C++ comment begins with the comment delimiter // and is terminated by the end of a line. C-style comments are also allowed. The first (noncomment) line of the function body declares three floating point variables a, b, and c and the next statement reads the values from the input. The C++ input/output library defines cin and associates it with standard input. The >> operator extracts a value from an input source, here cin, and puts the value into a variable; as shown, >> operators can be strung together to read more than one item. Specifically, three numbers are extracted from the input and put into a, b, and c. This kind of input is analogous to using C's fscanf function, except that no format string is specified.

Although C's right-shift operator is being used for input here, it can also still be used to do shifts. C++ operators, including >>, work like their C counterparts with operands of built-in type—int, float, etc.—but can be given new meanings when used with other kinds of operands. As we will see in Chapter 4, C++ allows programmers to define new types using classes. The variable cin has such a type, and the class gives >> its input meaning.

The third line of the function body prints the values of the variables. The C++ input/output library defines cout and associates it with standard output. The << operator inserts output into an output sink, here cout. As shown, << operators can be strung together to write more than one item. The statement in our example writes the value of a, then a character string consisting of a comma and a space, then the value of b, and so on. The C++ input/output library provides endl for ending a line. It writes a new-line character and flushes any buffer associated with the output sink (see Notes and Comments 3.1). More generally, input and output operations can be controlled by a variety of *manipulators*, like endl, provided by C++'s input/output library.

3.2 Variables, Objects, and Types

From a microscopic viewpoint, a computer program specifies a procedure for altering the values of objects using symbolic references to the objects, called variables. An *object* is an area of the computer's memory that has a *value*, a particular

bit pattern, stored in it. The *type* of an object specifies how the bits stored as the object's value are to be manipulated. A *variable* is an association between a name and an object; the type of a variable specifies the type of objects that the variable can be associated with.

Let's look at a simple example. The code

ch3/simplecpp.C

```
int i = 3;
float x = 10.0;
```

declares two variables and associates them with objects that have particular values. We illustrate this schematically as

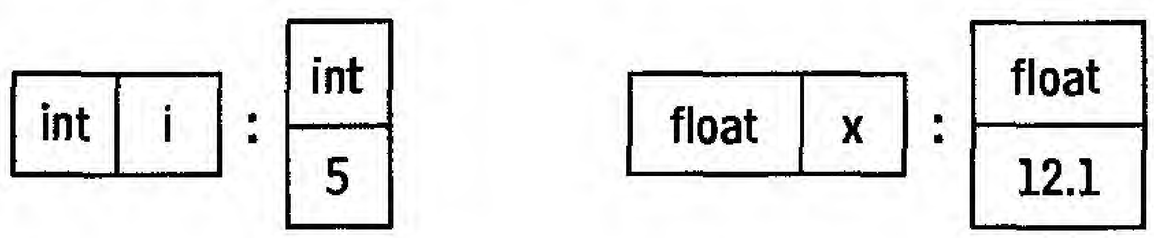
Each variable-object association is represented by two boxes separated by a colon: The variable is to the left and the object is to the right of the colon. Variable boxes are split into two parts giving the variable's type and name. Likewise object boxes are each divided into two smaller boxes, the top one containing the object's type and the bottom one containing the object's value. In the preceding example, the variable named i has type int and is associated with an object of type int with value 3; the variable named x has type float and is associated with an object of type float having value 10.0.

Executing the code

ch3/simplecpp.C

```
i = 5;
x = 12.1;
```

changes the values of the objects, giving



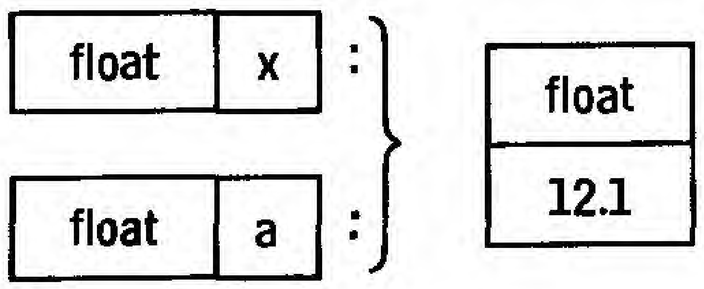
The assignment statements have changed the value of each object but not the types or the association between names and objects.

C++ extends C by allowing new associations to be created between variables and existing objects. The code

ch3/simplecpp.C

```
float x = 12.1;
float& a = x;
```

creates an association between the variable a, called a *reference*, and the object referred to by a. In pictures it creates



The variables x and a both access the same bits in the same way. References are discussed in more detail in Section 3.9.

3.3 C++ Built-In Types and Operations

C++ provides built-in (i.e., predefined or fundamental) types, operations, type-checking rules, and type conversion rules that are almost identical to those of C. We discuss the few differences in this section. More advanced aspects of C++'s type system with no C analog, including the ability to define new types, are described beginning in Chapter 4.

3.3.1 Characters

A C++ char is an integral type large enough to hold a single character. In C, the type of a character constant, like 'x', is int; in C++, the type of a character constant is char. The most important implication of this is that programs can reliably distinguish between characters and integers. For example, the code

ch3/char-out.C

```
cout << 'a';
```

writes a to the output stream; if the C rule for character constants held in C++, the code would write a number, not a character. The sizeof operator also yields different results in the two languages: In C++, sizeof('x') has the value 1, but in C it has the same value as sizeof(int).

3.3.2 Enumerations

Both C and C++ provide enumerations for identifying the members of small sets with integer codes. In C, enumerations do not introduce a new type and the enumeration constants are integers. For example, in C the code

ch3/enum.C

```
enum Color { red, orange, yellow, green, blue, indigo, violet };
```

defines seven enumeration constants with values from 0 (red) to 6 (violet). Variables meant to hold these enumeration constants can be declared like this:

ch3/enum.C

```
enum Color phosphor_type;
```

Appearance to the contrary, this declares phosphor_type to be a variable with an integral type; the particular integral type used is chosen by the compiler. Any integral value can be assigned to phosphor_type.

In C++, however, enumerations introduce a new type distinct from all existing types. Thus

ch3/enum.C

```
Color c = green;
```

declares the variable c to be of type Color with initial value green. As in C, C++ initializes each name in the *enumerator list* with an integer number code, starting with 0 and increasing by 1 going left to right. You may specify the integer values to be associated with each identifier in the enumerator list like this:

ch3/enum.C

```
enum Polygon { triangle = 3, quadrilateral = 4, pentagon = 5 };
```

Enumeration values can convert to ints when they are used as arithmetic and relational operands. For example, the expression c <= yellow yields 1 when c has any of the values red, orange, or yellow, and 0 otherwise. But ints are not convertible to enumerations: In Polygon p = triangle + 1, the sum of triangle and 1 gives the int 4, legally, but the int cannot be assigned to p.

3.4 Operator Precedence and Associativity

As in C, the interpretation of any C++ expression is determined by the *precedence* and *associativity* of the operators in the expression. Each C++ operator has a precedence: Operators in an expression are evaluated in order of highest to lowest precedence. C++'s operator precedence and associativity rules are the same as those for C, except for the addition of operators that are in C++ but not in C: ::, new, delete, ->*, and .*. The precedences and associativities of all C++ operators are given in Table 2.7 on page 31.

3.5 Input and Output

Section 3.1 introduced enough of the ideas of C++ I/O to read and write simple strings and numbers. This section describes some additional capabilities that are commonly needed and some aspects of compatibility with C and FORTRAN I/O.

Like C I/O, the iostream facilities of C++ are based on the notion of *streams*. A stream is a sequence of bytes that are produced (output) or consumed (input) by a program. Usually a stream is associated with a file or a device, like a terminal; in some systems, a stream can also be associated with a pipe, a mechanism for communication among programs. Regardless, a stream is a place either to get or

put bytes, and the C++ iostream system is a way of moving external data to and from objects.

The three streams cin, cout, and cerr, corresponding to the program's standard input, standard output, and standard error as defined by the operating system, are available automatically. On most systems, input from cin comes from the keyboard and output to cout and cerr goes to the screen. Operating system facilities can associate standard input, standard output, and standard error to, for example, files or other programs.

As data are read from an input stream, the bytes in the stream are interpreted according to the kind of input that is requested. The >> operator having a stream as its left operand requests that the bytes in the stream be interpreted as characters. If the variable x is a number, then the statement

ch3/iodemo.C

```
cin >> x;
```

reads characters from cin, converts them into a number of the appropriate type, and stores it into x. Blanks, newline characters, and tab characters, all called *white-space*, separate data items read by the >> operator; whitespace is otherwise ignored. For convenience, >> operators can be concatenated to read multiple data items:

ch3/iodemo.C

```
cin >> x >> y;
```

Contrast this with the analogous C code (which also works in C++):

ch3/input.c

```
float x;
/* ... */
fscanf(stdin, "%f", &x);
```

In the C code, the specification in the format string must be consistent with the variable to be stored into, and the address of the variable must be passed. Make a mistake in either and you might get a core dump. In the C++ code, the type of x dictates what is to be read, the variable x dictates where the result is to be stored, and no careless mistake will yield a core dump. Although C's stdio.h is available and works in C++, we recommend the following:

- Prefer the I/O facilities of iostream.h over those of stdio.h.

Output is done with the << operator:

ch3/iodemo.C

```
cout << x;
```

This causes the value of the variable x to be formatted into a sequence of characters and written to cout. No whitespace is inserted, so

ch3/iodemo.C

```
int i = 1;
int j = 2;
cout << i << j;
```

writes 12, probably not what you intended.

- Insert your own whitespace on output.

Whitespace can be inserted by writing character strings to the output. The previous example is correctly written as:

ch3/iodemo.C

```
int i = 1;
int j = 2;
cout << i << " " << j;
```

which writes 1 2.You will probably also want to split your output into multiple lines. A new line can be begun by writing the endl manipulator, which ends a line:

ch3/iodemo.C

```
int i = 1;
int j = 2;
cout << i << endl << j << endl;
```

This produces the output

```
1
2
```

Of course, it is sometimes necessary to have more control over input and output than the preceding simple examples illustrate. For example, many FORTRAN programs expect input that is organized into fixed columns; most C++ programs do not produce fixed-column output. Hence writing a C++ program that produces output for later consumption by a FORTRAN program requires altering the behavior of C++ streams. In particular, C++ uses a lowercase e in exponential notation, and it omits the decimal point in floating point output if the number has an integral value (e.g., 20.0 is written as 20). This can be changed. For example, here is a C++ program that writes three numbers that can be read by a FORTRAN program with FORMAT(2G15.8,F10.3):

ch3/fortran-compatible-io.C

```
#include <iostream.h>
#include <iomanip.h>

int main() {
    double a = 3.14159;
    double b = 1 / a;
    double c = 10 * a;
```

```
    // Use FORTRAN compatibility output.
    cout << setiosflags(ios::showpoint | ios::uppercase);

    // Write data in G15.8 format.
    cout << setw(15) << setprecision(8) << a;
    cout << setw(15) << setprecision(8) << b;

    // Write in F10.3 format
    cout << setiosflags(ios::fixed);
    cout << setw(10) << setprecision(3) << c << endl;

    return 0;
}
```

The formatting behavior of cout is set with three manipulators: setw for field width, setprecision for field precision, and setiosflags for various control flags. Inserting setw and setprecision into cout with the << operator affects the formatting of the next item inserted into cout. The manipulator setiosflags takes various flags that determine, for example, whether a decimal point is always shown (ios::showpoint) and the case of the E in scientific notation (ios::uppercase). To write in fixed format (i.e., F instead of G), we again set a flag: ios::fixed. To use these manipulators, the header file iomanip.h is required in addition to iostream.h. Other manipulators are available for controlling formatting; refer to your system's documentation or to [111] for details.

Besides controlling formatting, we also need to be able to test for end-of-file and error conditions. Using stdio.h, we simply test the value returned by fscanf() or fprintf(). With iostream.h, each stream has a *state*, which can be tested by testing the stream itself. For example, the following code fragment reads numbers and prints a table of their square roots:

ch3/sqrtTable.C

```
float x;
while (cin >> x) {
    cout << setw(25) << x << setw(25) << sqrt(x) << endl;
}
```

The expression cin >> x sets x from a value read from cin; the result of the expression is the stream cin; if it is in the good state, meaning that neither an error nor end-of-file has occured, testing it yields true; otherwise the test yields false. Thus the loop stops at the end of the input file. The output statement is executed exactly once per input item, even if there are no input items (in which case the output statement is not executed). Other checks of stream state are available; refer to your system's documentation or to [111] for details.

I/O to files is also accomplished via streams in C++. An output stream is connected to a file using an object of type ofstream, like this:

ch3/io.C

```
#include <iostream.h>
#include <fstream.h>

ofstream out("pi.out");
out << 3.14159 << endl;
```

The variable out is of type ofstream ("output file stream") and can be used just like cout. It is connected to the file named pi.out.

Similarly, input file streams are connected to files via objects of type ifstream, and files used for both reading and writing are objects of type fstream. To use these types, you must include the header fstream.h.

Finally, C++ also provides analogs of C's sscanf() and sprintf() functions for converting numerical data to its text representation and vice versa, again via streams. For details, refer to [111].

3.6 Declarations

C++ declarations are essentially C declarations, with one important difference: C++ declarations are statements. Since declarations are statements, they can be intermixed with other statements: They need not—and should not—be grouped together and placed before the "executable" statements of a function as is required in C. We suggest that you

- Use one declaration statement for each variable; declare each variable just before its first use.

One-variable declarations are easier to read and edit than multiple-variable declarations; declaring variables near their use makes their type easier to find and edit.

Formulas that contain physical constants, such as Planck's constant, are usually written as a symbol, not as a number. This helps us recognize the physical significance of the formula, takes less time to write, avoids transcription errors, and allows for the occasional change in a physical constant. Similarly, C++ *constant objects* are symbols for objects with a fixed value. In traditional C, such constants are specified with the macro preprocessor. For example,

ch3/define.c

```
#define H 6.6256E-34
```

The preprocessor replaces the name H with the constant 6.6256E-34 before the compiler sees the code. Therefore a source-level debugger won't display a variable named H, because there won't be one.

Both ANSI C and C++ provide a solution: Declare a const variable by using the const type modifier just preceding the type name:

ch3/const.C

```
const float h = 6.6256e-34;    // Planck's constant (mks units)
```

This declares h to be an immutable floating point constant. The symbol h can now appear in computations or be printed, but its value can't be changed. (Despite the common use in physics of single character symbols for physical constants, it is a bad idea to use such a symbol as the name of a constant that is known to large parts of a program. Thus h_Planck would be a better choice for this constant.)

■ Create symbolic constants by declaring const variables, not by using #define.

An arbitrary expression can be used to initialize a const variable. The word constant does not mean that the value has to be known before the program is run. Rather it means that once the variable is initialized, it can't be changed. The following code is correct:

ch3/const.C

```
void f() {
    float x;
    cin >> x;              // Read value of x
    const float xc = x;
    // ...
}
```

3.7 Pointers

C++ pointers are essentially ANSI C pointers. ANSI C extends the pointers of traditional C with const pointers and pointers to const objects. If you are not familiar with these extensions, read Section 2.12, as the extensions are important in C++.

The major practical difference between pointers in ANSI C and in C++ relates to null pointers. Both C and C++ have a null pointer value that points explicitly to no object or function. In both languages, the integer constant 0 may be converted to any pointer type, with the result guaranteed to be the null pointer. (The null pointer may or may not be represented by a word with all zero bits, and the null pointer may have different representations for different type pointers.) In ANSI C, the null pointer is conventionally written as NULL. NULL is defined by a macro in the standard header file stddef.h. For technical reasons (explained in Notes and Comments 3.2), the definition often supplied there is not appropriate for use in C++ programs. Since the ANSI C header files are shared between C and C++, we recommend the following:

■ In C++, use the integer constant 0 for the null pointer.

Finally, we adopt a different style of declaring pointers than is typically used by C programmers. The formal syntax of pointer declarators associates the * with the name of the variable being declared. In fact, C programmers often write

ch3/ptrdclstyles.C

```
int *p;
```

instead of the

ch3/ptrdclstyles.C

```
int* p;
```

that we write. Our style, which is also used in some other C++ books, seems more natural: p is an int*, a pointer to an int. The C style mirrors the use of p in an expression: The expression *p has type int. The C style is preferable if you declare several variables in one declaration statement, like this:

ch3/ptrdclstyles.C

```
int* p1, p2;
```

Here p1 is a pointer to an int, but p2 is an int, not a pointer to an int. We recommend avoiding this pitfall:

- Declare one variable per statement.

Regardless of how you write pointer declarators, "the declarator syntax of C – and therefore of C++ – is widely recognized as being unnecessarily hard to read and write" [44, p. 132].

3.8 Memory Management

C programs allocate and deallocate memory using the functions malloc() and free(). These are replaced in C++ with the operators new and delete, respectively. A C program might allocate and deallocate a float on the heap like this:

ch3/heap.c

```
float *fp = (float *) malloc(sizeof(float));    /* Allocate a float on the heap. */
/* ... */
free(fp);    /* Deallocate. */
```

The equivalent C++ program would be written like this:

ch3/new.C

```
float* fp = new float;    // Allocate a float on the heap.
// ...
delete fp;    // Deallocate.
```

The new operator takes a type as its operand and allocates an object of that type on the heap, returning a pointer to the object. If new is unable to allocate the necessary memory, its default action is to return a null pointer. The C++ form is more succinct and less error prone than the C form. No type cast is needed. Moreover, new

and delete perform additional functions when used with the programmer-defined types we introduce in Chapter 4.

- Use new and delete for memory management, not malloc() and free().

Special forms of the new and delete operators are used to allocate and deallocate arrays of objects on the heap. A C programmer might allocate an array of n floats like this:

ch3/heap.c

```
int n;
float *x;
scanf("%d", &n);     /* Read count. */
x = (float *) malloc(n * sizeof(float));
/* ... */
free(x);
```

The C++ programmer would write

ch3/new.C

```
int n;
cin >> n;     // Read count
float* x = new float[n];
// ...
delete [] x;
```

The square brackets ([ ]) are used to supply the number of elements to new and to indicate to delete that an array of objects is being deallocated. Note that the count is not given when deallocating an array of objects.

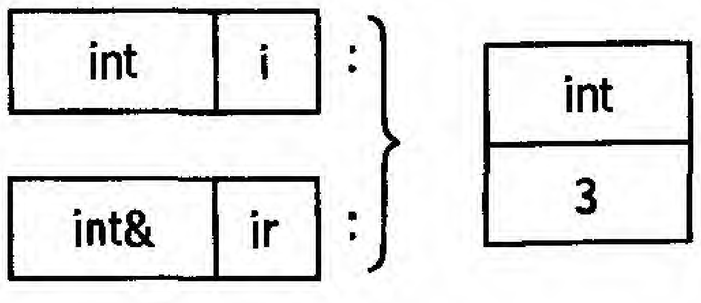
3.9 References

A *reference* is an alias for an object of a specified type. For example, the declarations

ch3/refs.C

```
int i = 3;        // Variable
int& ir = i;      // Reference to variable i
```

declare a variable, i, and a reference to it, ir. The second declaration declares ir to be of type int& , which means "reference to int." A reference introduces a new name for an existing object, so that after the preceding code we have



The connection between a reference and an object is formed when the reference is initialized; hence references must always be initialized with some object, and they refer thereafter to that object.

Given the preceding declarations of i and ir, executing

ch3/refs.C

```
cout << ir << endl;
ir = 4;
```

prints 3 and then changes the value of i to 4. Now executing

ch3/refs.C

```
cout << ir << endl;
```

prints 4. Notice that assigning to a reference alters the object referred to, not the reference itself.

Both pointers and references provide access to an object, but they are different in important ways. A pointer points to an object, the pointer can be changed to point to a different object, the indirection operator must be used to obtain the object, and storage is used for the pointer's value. A reference acts like another name for the object, it can point only at one object, no operator is needed for access, and no storage need be allocated.

The general form of a reference declaration is a type followed by an ampersand (&), a name, and an initialization. If the type is, for example, float, then the reference type will be float&. Thus float& fr declares the name fr to be a variable of type float& read as reference to float or float reference. As with pointers, there are also references to const objects. For example, in

ch3/refs.C

```
const float pi = 3.14159;
const float& unit_circle_area = pi;
```

unit_circle_area is a const reference to the constant object pi. It is also possible to have a const reference to a non-const object:

ch3/refs.C

```
float x;
const float& xr = x;
x = 5.0;
```

The compiler will flag any attempt to alter the value of x by assigning to xr, but it will allow assignment to x itself.

References allow C++ function arguments to be passed by reference, as well as the C-like pass by value, without needing pointers. In addition, function calls can appear on the left-hand side of assignments by returning references. An example appears in Section 5.4.

3.10 Functions

C++ functions are almost identical to ANSI C functions in prototype form. (If you are not familiar with ANSI C function prototypes and function definitions in prototype form, read Section 2.16 instead of this section.) There is a minor difference in the syntax of the prototype of a function with no arguments, and it is possible to pass arguments by reference. We look at each of these differences in turn.

3.10.1 Prototypes

In ANSI C, the prototype for a function with no arguments is written like this:

ch3/void.C

```
void f(void);
```

C++ also permits the prototype to look like this:

ch3/void.C

```
void f();
```

We prefer the second form and use it throughout the book. People who program in both C and C++ sometimes prefer the first form because in ANSI C the second form means that argument types are not checked.

3.10.2 Arguments

In C, all function arguments are passed by value; a function that needs to alter its arguments must be passed pointers to those arguments and the objects must be accessed through those pointers. In C++, references can be used to achieve the same effect more directly. Understanding how this can be done requires understanding how arguments are passed.

The arguments that appear in the function definition are called *formal arguments*, and the arguments passed to the function are called *actual arguments*. Each formal argument is a local variable of the function, and each formal argument is initialized with its corresponding actual argument. For example,

ch3/funcarg.C

```
void f(int i, float x, float* a) {
    i = 100;
    x = 101.0;
    a[0] = 0.0;
}
```

declares three formal arguments: i, x, and a. Calling the function like this

ch3/funcarg.C

```
int j = 1;
int k = 2;
float y[] = {3.0, 4.0, 5.0};
// ...
f(j, k, y);
```

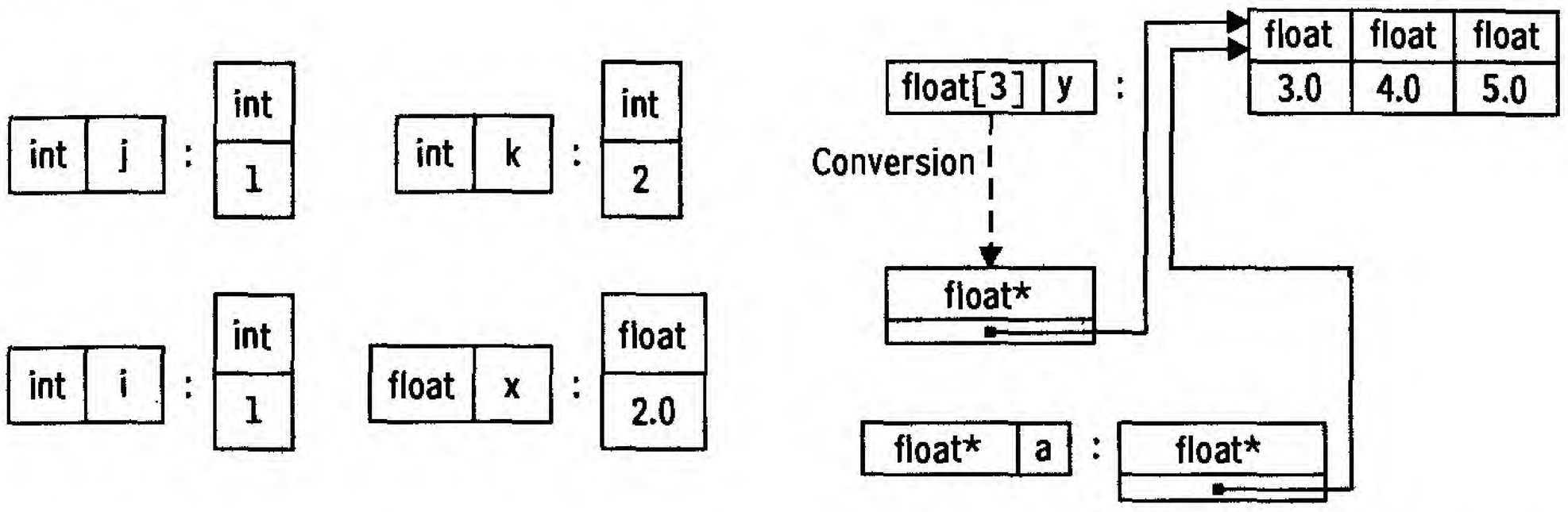
causes actions equivalent to

ch3/funcarg.C

```
int i = j;
float x = k;
float* a = y;

i = 100;
x = 101.0;
a[0] = 0.0;
```

First the three formal arguments are initialized from their corresponding actual arguments. To initialize x, the value of k is converted from an int to a float; to initialize a, a float* value that is equal to the address of the first element of y is used. After this initialization, the association between variables and objects looks like this:



In this example, the assignments to i and x alter those variables within the f() function, but have no effect on the values of the actual arguments j and k in the calling function; the arguments are said to be *passed by value*.

To alter an argument's value, or to avoid copying larger nonarray objects such as structures, reference formal arguments can be used:

ch3/funcarg.C

```
void swap(int& i1, int& i2) {
    // Swap arguments
    int temp = i1;
    i1 = i2;
    i2 = temp;
}
```

The formal arguments i1 and i2 are references (cf. Section 3.9). Initializing them with the actual arguments forms an association between the formal argument names and the corresponding actual argument objects. Then changes to the formal arguments affect the actual arguments; this is called *call by reference*.

A more detailed discussion of function arguments appears in Section 5.3.

3.10.3 Inline Functions

For most functions, the time required to execute the function's body dominates the total time taken by a call to the function. In other words, the overhead of calling the function is insignificant compared to the computation done by the function. Occasionally, however, the overhead is a significant fraction of the total function call cost. If such a function is called many times, the overhead can be a significant fraction of a program's total execution time.

When function call overhead is significant, a function definition can be preceded by the keyword inline. This instructs the compiler to attempt to generate inline code for calls to the function. Here is an example:

ch3/doubleSqr.C

```
inline double sqr(double x) {
    return x * x;
}
```

When this function is called, we expect the compiler to generate inline code without function call overhead. The inline keyword is a suggestion to the compiler; a C++ compiler is not required to obey the suggestion. Any decent compiler will generate inline code for a function like sqr(). There is, however, considerable variation among compilers in how they handle more complex inline functions, especially those containing loops.

The definition of an inline function must have been seen by the compiler before the function is called. It is not enough for the compiler to have seen a function prototype; without the definition itself, the compiler cannot generate inline code. Thus an inline function definition is usually put in a header file, in place of the function prototype.

Although inline functions eliminate function call overhead, they can introduce other overheads. When a function is inlined, its code is duplicated for each call. Excessive use of inline may result in large programs. Large programs may have poor instruction cache behavior or cause excessive paging in virtual memory systems.

Overusing inline functions can also lengthen compile and link times: When the definition of an inline function is changed, every function that calls it must be recompiled; when the body of a noninline function is changed, only that function need be recompiled. Moreover, the definitions of inline functions may require other header files, and changes in those files may cause recompilation of the caller of an inline function. This leads us to recommend the following:

- Use inline functions sparingly.

On the other hand, not inlining small functions that do little computation can make programs both bigger and slower. To complicate matters, the decision is somewhat machine and compiler dependent.

One approach is not to inline any functions until the program runs and its performance can be measured. Functions that are performance bottlenecks can then be inlined. But one must understand what is being measured. Take the extreme in which a function f() repeatedly calls a function g() with an empty body: g() will not appear to take any time, yet making it inline could greatly improve the running time of f(). In deciding which functions to inline, it is important to consider both the time taken by the candidate function and the time taken by functions that call it.

Another factor is ease of debugging. On some systems, debugging an inline function is difficult because the function doesn't exist at runtime. On such systems, it might be appropriate not to inline any functions until the entire program is running.

We subscribe to a modified measure-before-inlining approach. We generally inline functions that just return a built-in type or a reference. We also generally inline functions that just return the result of calling another function. We sometimes inline functions that do a simple computation and are likely to be called from within loops in other functions. Beyond this, we measure first.

3.11 Notes and Comments

3.1 The effect of writing endl to an output stream in C++ is the same as writing a newline character to a C stream that has been put in line-buffering mode by a call to setbuf() with a buffer mode of _IOLBF.

3.2 In C, the standard header file stddef.h defines NULL, often as ((void *) 0). Since C allows an implicit conversion from a void* to any other pointer type, this value can be used in any situation that calls for a pointer. However, C++ does not allow implicit conversion from void* to other pointer types, implying that this definition of NULL is not suitable for use in C++.

3.3 Consistent indentation aids code readability. We have chosen one style, but there are other possibilities, each with its own adherents. One popular style aligns braces vertically:

ch3/vertstyle.C

```
if (a < b || b > 10)
    {
    a = 3;
    b = 4;
    }
```

This style shows the levels of nesting more clearly at the expense of fitting less code and thus less context in a given space. This was the deciding factor in choosing a code style for this book. A thoughtful discussion of indentation styles appears in [104, Chapter 6].

3.12 Exercises

3.1 In file regurgitate.C (page 67), change the declaration of a, b, and c to type int. Does the program still work? Rewrite the original program in C using scanf() and printf(), and then make the same change. Does the program still work?

3.2 A polynomial

$$P(x) = \sum_{i=0}^{n-1} a_i x^i$$

should be evaluated using Horner's method:

$$P(x) = (\cdots((a_{n-1}x + a_{n-2})x + a_{n-3})x + \cdots)x + a_0.$$

Write a program that reads from cin the number of coefficients, n, followed by n coefficients, followed by an arbitrary number of x values. For each x value, compute and print $P(x)$.

3.3 What do the following statements mean?

ch3/ex-refs.C

```
int i = 5;
int& ir = i;
int* ip = &ir;
int*& ipr = ip;
const int& cir = i;
```

CHAPTER 4

Classes

SCIENTIFIC and engineering problems are rarely posed directly in terms of the computer's intrinsic types: bits, bytes, integers, and floating point numbers. C++ addresses this problem by providing a mechanism—*classes*—for defining new types and operations for manipulating objects of those types. Classes have many roles in C++ programming: To write object-oriented programs in C++ is to write and use classes.

This chapter introduces the basics of classes, focusing on using them to define new types. The first section uses classes to define two simple types, Point and Line. The second section uses a class to define a type that is an array of floating point numbers; we then see how to generalize this class into an array type capable of holding elements of arbitrary type. We then discuss *exceptions*, a means of reporting error conditions that uses classes. The chapter concludes with a discussion of nested classes, a mechanism for grouping related classes.

4.1 Two Simple Classes

In this section, we develop two simple example classes, Point and Line, each representing a geometric object in two-dimensional space. To focus on classes, we ignore the potential numerical errors that can occur in geometric computing.

4.1.1 Classes Are Programmer-Defined Types

A class defines a new type, defining both the representation and the operations for objects of the new type. A *class declaration* consists of the keyword class followed by a name:

class *class-name*;

This simply states that *class-name* is a type. A *class definition* begins with a class declaration but adds declarations of variables to define the type's representation and declarations of functions to define the type's behavior. (See Notes and Comments 4.1 for more on the distinction between declaration and definition.)

The variables are called *data members*, and the functions are called *member functions*. Collectively, the data members and member functions are called the class's *members*.

A class definition consists of the keyword class followed by the class's name, followed by a *class-body* enclosed in braces, all terminated by a semicolon:

```
class class-name {
    class-body
};
```

Since the closing braces found in other contexts are not followed by a semicolon,

- Remember the trailing semicolon on class definitions.

The *class-body* consists of definitions of the class's members.

Once defined, a class can be used like a built-in type:

ch4/Point0.C

```
class Point {
    // ... class body ...
};

Point a;
Point square_vertices[4];
```

Executing the first variable declaration creates an object of type Point and associates it with the variable a; executing the second declaration creates an array of four Point objects and associates it with the variable square_vertices. Each of these five Point objects is an *instance* of the class Point. An object that is an instance of a class (as distinguished from an object of a built-in type) is called a *class object*.

4.1.2 Member Functions and Member Variables

Now let's design and implement our two classes, starting with Point. A class's member functions determine the behavior of its instances. Let's assume that the desired behaviors of our Point and Line classes are these:

Point	Line
Create from coordinates.	Create from two points.
Compute distance to another point.	Create from point and direction.
Compute distance to a line.	Intersect with another line.
Get x- and y-coordinates.	Compute distance to a point.

We begin with Point:

ch4/Point.h

```
typedef float Number;

class Line;

class Point {
public:
    Point();                          // Create uninitialized
    Point(Number x, Number y);        // Create from (x,y)
    Number distance(Point point);     // Distance to another point
    Number distance(Line line);       // Distance to a line
    Number x();                       // Get x-coordinate
    Number y();                       // Get y-coordinate
private:
    Number the_x;                     // x-coordinate
    Number the_y;                     // y-coordinate
};
```

The typedef makes Number a synonym for float. Writing declarations with Number instead of float makes it easy to change later to double precison: Change float to double in the typedef and recompile. The next statement declares that Line is a class but does not give any information about the class. This is enough, however, to let us use the word Line as a type even before we've said what a Line is. Such a declaration is called a *forward declaration*. Note that function prototypes are another kind of forward declaration, allowing us to call functions without first providing a complete function definition.

Point's class body declares six member functions that correspond to the preceding list. The keyword public indicates that the members declared following it are accessible from outside of the class. The keyword private indicates the opposite, namely that the members following it are not accessible from outside of the class. The data members the_x and the_y will hold the coordinates of each point. We first look at how Point objects are used and then return (page 89) to the role of the data members.

4.1.3 Calling Member Functions

Knowing the member functions of Point, we can consider an example of its use:

ch4/distance.C

```
#include <iostream.h>
#include "ch4/Point.h"
```

```
int main() {
    // Read the coordinates of a point and print the distance
    // from the origin to the point.
    Point origin(0, 0);

    Number x;
    Number y;
    cin >> x >> y;
    Point p(x, y);

    cout << origin.distance(p) << endl;
    return 0;
}
```

The program includes the header file iostream.h, making input and output available. The header file containing the definition of Point is included so that we can use our newly defined Point type.

The first statement in the body of main() declares origin to be a variable of type Point and initializes it to have coordinates (0, 0). A class object is created by calling a *constructor* member function. Constructors are distinguished syntactically from other member functions by having the name of their class and no return type. Point has two constructors, one that does not take arguments and one that takes the coordinates of the point being created. Since the declaration of origin provides coordinates, the second constructor is called to create and initialize the object for origin.

Notice that we did not call the constructor explicitly; we declared a Point variable; the compiler arranged for appropriate memory to be allocated and for the constructor to be called. The arguments in parentheses following the name of the variable are passed to the constructor function, which typically uses them to initialize the class object being created.

Next the variables x and y are declared and their values read from cin. These values are passed to the constructor when the variable p is declared.

The program concludes by calculating the distance and writing the result to cout. The expression origin.distance(p) calls the member function distance() to compute the distance between origin and p. A member function (other than a constructor) is called with a class object, in this case the Point object origin. The class object is followed by the *class member access operator*, ., the "dot" operator. Then comes the member function name, here distance, and its arguments, here p.

Point has two member functions named distance. You distinguish between them because they require different argument types. C++ distinguishes between them in the same way. Thus origin.distance(p) computes the distance between two

points by calling the Point member function distance(Point); if a Line object had been supplied as an argument, then distance(Line) would have been called to compute the distance between a point and a line. This is similar to the operation of generic intrinsic functions in FORTRAN, the difference being that in C++, programmer-defined functions may be selected based on argument types.

Member functions can also be called with a pointer to a class object:

ch4/ptrselect.C

```
Point p1(1, 2);
Point* p_ptr;
// ... set p_ptr ...
cout << p_ptr->distance(p1) << endl;
```

Here the member function distance() is being called with the Point object pointed to by p_ptr. The -> operator, also called the class member access operator, selects a member of the class object pointed to by its left operand.

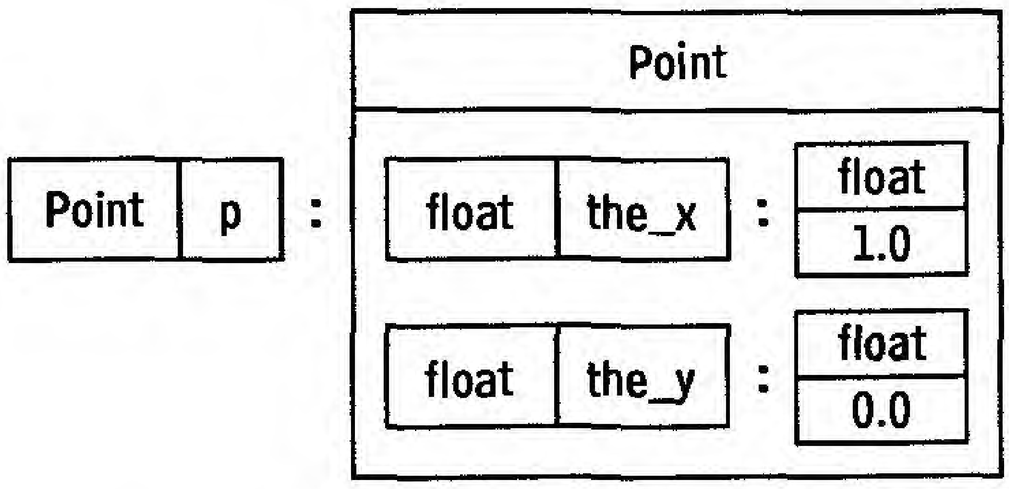
4.1.4 Member Variables

We chose to represent each Point object by a pair of coordinates. Thus there are two data members declared in the Point class (cf. page 87). Data member declarations have the same form and follow the same rules as ordinary declarations except that no initializer can be supplied; they declare data members because they are contained in the class body. The Point class body contains two variable declarations, so every instance of Point contains two Number objects, which are referred to within the class by the names the_x and the_y.

The keyword private states that the member data are *private data members.* Private data members can be accessed only by one of the class's own member functions. (There is an exception to this rule: A class can explicitly grant access to other functions. See Section 6.10.) This restricted access limits the code that must be modified if we modify the storage representation of Point. We chose a Cartesian representation this time. If we discover later that radius and angle would be a better representation for a Point, we need only change Point and its members; code that uses Point will not be affected.

- Data members of a class should be declared private to ensure that they are only accessed from within member functions.

Point is a type, and the relation between variables and objects for this type is the same as the relation between variables and objects for built-in types. However, the value content of a class object is composed of the value content of the object's members. In terms of the diagrams introduced in Chapters 3 and 2, a Point object declared Point p(1.0,0.0) can be sketched as



The storage size of p will likely be 2*sizeof(float), and a program could use many such objects during computation.

4.1.5 Class Line

Now let's look at the definition of Line:

ch4/Line.h

```
class Line {
public:
    Line(Point p1, Point p2);                  // Line through two points
    Line(Point p, Number xDir, Number yDir);   // Line through p with tangent <xDir, yDir>
    Point intersect(Line line);                // Intersection with line
    Number distance(Point point);              // Distance to a point

    static Number parallelism_tolerance;       // Smallest nonparallel angle (radians)
private:
    // ax + by + c = 0
    Number a;
    Number b;
    Number c;
};
```

We represent each line by the equation $ax + by + c = 0$; the three data members hold the equation's coefficients. The parallelism_tolerance datum will be discussed in Section 4.1.7.

The code

ch4/intersect.C

```
Line l1(Point(0,0), Point(0,1));      // Vertical line through origin
Line l2(Point(1,2), 1, 1);            // Line through (1,2) in direction <1,1>

Point intersection;
intersection = l1.intersect(l2);
cout << "(" << intersection.x() << "," << intersection.y() << ")" << endl;
```

creates two Line objects and then computes and prints the coordinates of their intersection. The first two statements each declare a variable of type Line, initialized by one of the two Line constructors, with the choice of constructor dictated by the arguments provided. The third statement declares a Point variable; since no constructor arguments are provided, the Point constructor having no arguments is used. The expression l1.intersect(l2) returns a Point object, which is then assigned to the intersection Point variable. (Assignment of class objects of the same type is, by default, equivalent to assigning corresponding data members.) Finally, the coordinates of the point of intersection are obtained by calling the x() and y() member functions, and the results are printed.

4.1.6 Member Function Definition

Having looked at the Point and Line classes from the perspective of their users, we now turn to implementing their behaviors by defining the member functions. Let's start with the Point constructors:

ch4/Point.C

```
Point::Point(Number x, Number y) {
    the_x = x;
    the_y = y;
}

Point::Point() {
}
```

The notation Point::Point says that Point is a member function of the class Point. (The left operand of the *scope resolution operator* :: is a class name, and the right operand is a member name.) Since the names of the first two functions are the class name, these two functions are constructors. The first one is called when a Point object is created and coordinates are supplied; the second one is called when no coordinates are supplied. For example, when the declaration

ch4/tPoint.C

```
Point p1(3.0, 4.0);
```

is executed, C++ creates a Point object, associates it with the variable p1, and calls the first constructor with 3.0 and 4.0 as arguments. The Point object contains two subobjects the_x and the_y, which are set by the body of the constructor. When the declaration

ch4/tPoint.C

```
Point p2;
```

is executed, the second constructor is called since there are no arguments.

The x() and y() member functions are simple:

ch4/Point.C

```
Number Point::x() { return the_x; }
Number Point::y() { return the_y; }
```

When the expression p1.x() is evaluated, the member function x() is called for the Point object p1. Within a member function, data member names refer to the values for the particular object supplied with the function. Thus for the call p1.x(), the value in x() of the variable the_x is the value of p1's subobject the_x.

Here is another view of the connection between member functions and member data. Suppose a program declared 10 variables of type Point. There would be 10 Point objects, each of which would contain a Number object named the_x and a Number object named the_y. Thus there would be 10 the_x objects and 10 the_y objects, paired together and accessible only as components of Point objects. There would be one copy of the code for the member function Point::x(). When the member function x() is called for one of these objects, say p5, by p5.x(), then the value of the_x inside of Point::x() will be the value stored in the Point object p5.

Let's take a look at a member function that works simultaneously with two instances:

ch4/Point.C

```
Number Point::distance(Point point) {
    Number x_dif = the_x - point.the_x;
    Number y_dif = the_y - point.the_y;
    return sqrt(x_dif * x_dif + y_dif * y_dif);
}
```

Consider executing the expression p1.distance(origin). When the function is called, the contents of the origin object are copied into the local object point. The variables the_x and the_y refer to those members of p1. The expressions point.the_x and point.the_y refer to the corresponding members of the object point. Thus during execution of a member function, data member names refer to contents of the object used to select the member function and members of other instances can be accessed using the member selection or "dot" operator. Notice that distance() can access the private members of point because it is a member function of Point, the same class of which point is an instance.

Point::distance(Line) is implemented most easily by letting Line do the work:

ch4/Point.C

```
Number Point::distance(Line line) { return line.distance(*this); }
```

The local variable this is defined automatically by C++ for each (non-static) member function. It is a pointer—here of type Point*—that points to the object used to select the member function. For example, if p1.distance(line1) is being evaluated,

this points to p1 while distance() is executing. Then inside Point::distance(Line), the expression line.distance(*this) calls Line's distance() member function for the Line object line and passes it the Point p1 as an argument.

The member functions of Line are simple (because we don't handle error conditions); work through them to be sure you understand the basics of classes:

ch4/Line.C

```
Line::Line(Point p1, Point p2) {
    // Construct a Line through points p1 and p2.
    if (p1.x() == p2.x()) {                    // Vertical line
        a = 1;
        b = 0;
        c = -p1.x();
    }
    else {                                     // Nonvertical line
        a = p2.y() - p1.y();
        b = p1.x() - p2.x();
        c = p1.y() * p2.x() - p2.y() * p1.x();
    }
}

Line::Line(Point p, Number xDir, Number yDir) {
    // Construct a Line through p with tangent <xDir, yDir>.
    a = yDir;
    b = -xDir;
    c = xDir * p.y() - yDir * p.x();
}

Number Line::distance(Point point) {
    // Returns the distance from point to the line.
    return abs( a * point.x() + b * point.y() + c ) / sqrt(a*a + b*b);
}
```

The implementation of intersect() uses parallelism_tolerance; we describe this variable and its use next.

4.1.7 Shared Objects inside Classes

Every Line object has its own private data: the values of a, b, and c that describe the particular line via the equation $ax + by + c = 0$. All Line objects may also need to share some data. For example, the tolerance level to avoid incorrect intersections of nearly parallel lines would be a value that all Line objects should share. If

we code the tolerance into the intersect() function, we will have to modify the function and rebuild the program to change the tolerance; if we store the tolerance as a member datum like a, b, or c, we waste storage.

C++ provides *static member data* for this kind of shared information. In our definition of class Line on page 90 we declared the member datum parallelism_tolerance to be a static Number value. The modifier static in member data causes C++ to associate the variable with the class Line but not to reserve storage for the data in each Line object.

The value of parallelism_tolerance can be used in the definition of intersect():

ch4/Line.C

```
Point Line::intersect(Line line) {
    // Intersect with line. If the angle between the two lines is less than
    // the parallelism_tolerance, return the point at infinity. The parallelism
    // test computes the square of the sin of the angle. We assume that the
    // tolerance is small enough for sin(theta) to be approximately equal to theta.
    Number det = a * line.b - line.a * b;
    Number sinsq = (det * det) / ( (a*a + b*b) * (line.a*line.a + line.b*line.b) );
    if (sinsq < parallelism_tolerance * parallelism_tolerance) {
        return Point(FLT_MAX, FLT_MAX);   // Point at infinity (FLT_MAX from float.h)
    }
    else {
        return Point( (b * line.c - line.b * c) / det, (c * line.a - line.c * a) / det );
    }
}
```

Each call to intersect() will compute the intersection of two particular lines, using the values of a, b, and c for those Lines. Every call to intersect() refers to the same parallelism_tolerance datum. Only one copy of this value exists no matter how many Line objects are created in the program.

This example uses a Number object, which is a float object in this program. However, a static member datum can be a built-in object or a class object.

The static member objects are given initial values outside the class body. The member names must be qualified with the scope resolution operator to connect the initialization back to the Line class:

ch4/Line.C

```
Number Line::parallelism_tolerance = .01745;   // 1 degree (in radians)
```

We used public static data in this example; see Exercise 4.8 for an examplé that illustrates the advantages of making even simple static data members private. Thus we can reset the value of the tolerance by assigning to the static member:

ch4/intersect.C

```
Line::parallelism_tolerance = .02618;   // 1.5 degrees (in radians)
```

Here we must again use the Line:: qualification, as we are outside of the class scope when we use the member.

In addition to static member data, C++ supports static member functions. Often these functions manipulate static member data. For example, Exercise 4.8 sets parallelism_tolerance with a function set_parallelism() declared with the modifier static; this function is called with the Line:: qualification in the same way that static member data are used.

4.2 An Array Class

Sections 2.13 and 3.8 show how runtime-sized one-dimensional arrays can be created, used, and deleted using pointers. Manipulating arrays as pointers is awkward and error prone. C++ provides the tools to create new array types by creating classes with array characteristics. This section introduces some of these tools, using a one-dimensional array of floating point numbers as an example. Later we revisit this example to generalize the class once we introduce more language features.

We shall call this version of our array SimpleFloatArray, and we want to define it so that it acts like we expect arrays to act. Thus we want code like the line-fitting function of Section 2.13 to be written like this:

ch4/linefit.C

```
#include <iostream.h>
#include "ch4/SimpleFloatArray.h"

void linefit() {
    // Create arrays with the desired number of elements
    int n;
    cin >> n;
    SimpleFloatArray x(n);
    SimpleFloatArray y(n);

    // Read the data points
    for (int i = 0; i < n; i++) {
        cin >> x[i] >> y[i];
    }

    // ... same as before ...
}
```

The requirements that such code places on the SimpleFloatArray include the following:

- Runtime sizing with size given as a constructor argument
- Access to element i in array x by evaluation of x[i]

- Automatic management of dynamically allocated memory associated with the array (no delete call needed).

In addition, we know from experience that we shall also want to

- Automatically copy array elements when arrays are copied
- Assign one array to another, and assign one float value to every array element
- Ask an array's size and resize an array dynamically.

These requirements arise when we use more than one array in a program or use arrays as data members of classes.

Each of these requirements leads to a member function. Most are special member functions that C++ applies either automatically or with a calling syntax different from either ordinary function call or member function call. Working down our list of requirements, we need the following member functions:

- `SimpleFloatArray(int n)`, a constructor taking the number of array elements to allocate. We have already seen that constructors are special functions called when objects are created; the call syntax places the construction arguments after the object name, as in `SimpleFloatArray x(n)`.
- `float& operator[](int i)`, an *operator* member function taking an integer subscript argument. Operator members are declared with the keyword `operator` followed by the operator symbol; in use the call syntax places the argument within the paired operator symbols, as in `x[i]`. We discuss the return type of this function when we define the operator.
- `~SimpleFloatArray()`, a *destructor* member function taking no arguments and having no return type. Destructors are declared by preceding the class name with a tilde, ~. C++ automatically calls a destructor when a variable associated with an object by a constructor call is destroyed (e.g., at the end of the linefit() function).
- `SimpleFloatArray(const SimpleFloatArray&)`, a *copy constructor* taking a reference to an object of the class as an argument. C++ calls this function when creating new objects from existing ones.
- `operator=(const SimpleFloatArray&)`, a *copy assignment operator* taking a reference to an object of the class as an argument. Again the `operator` keyword is used in the declaration, but only the operator is used to call the function. For example, `x=y` calls `x`'s `operator=()` member with an argument referring to `y`. Similarly, `operator=(float)` is called when `x=5.0` is executed.

- numElts() for number of elements and setSize(int) to reset the size. These are ordinary member functions, illustrating that SimpleFloatArray is a class with special member functions giving it array behavior, not a special class.

These member functions lead to a class definition like this:

ch4/SimpleFloatArray.h

```
class SimpleFloatArray {
public:
    SimpleFloatArray(int n);                                    // Create array of n elements
    SimpleFloatArray();                                         // Create array of 0 elements
    SimpleFloatArray(const SimpleFloatArray&);                  // Copy array
    ~SimpleFloatArray();                                        // Destroy array
    float& operator[ ](int i);                                  // Subscripting
    int numElts();                                              // Number of elements
    SimpleFloatArray& operator=(const SimpleFloatArray&);       // Array assignment
    SimpleFloatArray& operator=(float);                         // Scalar assignment
    void setSize(int n);                                        // Change size
private:
    int num_elts;                                               // Number of elements
    float* ptr_to_data;                                         // Pointer to built-in array
                                                                // of elements
    void copy(const SimpleFloatArray& a);                       // Copy in elements of a
};
```

In addition to the aforementioned member functions, we have a private member function copy() that copies elements from one array to another; it is called by both the copy constructor and the copy assignment operator, but it is not available to the class's users. We also have two member variables to hold the elements. The num_elts member remembers the number of elements, essential for copy and assignment operations. The ptr_to_data element points to the dynamically allocated data, exactly as we did in Section 2.13. This pointer is inaccessible to users of SimpleFloatArray; access is not needed because the operations of the class do all we want to do to arrays.

The operations of the class are coded in the member function definitions. The first constructor creates an n-element array by storing the element count and using new to allocate an n-element built-in array. The member variable ptr_to_data holds the pointer to the built-in array:

ch4/SimpleFloatArray.C

```
SimpleFloatArray::SimpleFloatArray(int n) {
    num_elts = n;
    ptr_to_data = new float[n];
}
```

```
SimpleFloatArray::SimpleFloatArray() {
    num_elts = 0;
    ptr_to_data = 0;
}
```

The second constructor creates an array with no elements: The count is set to zero and the pointer is set to the null pointer.

The third constructor creates a copy of the array passed to it. The copy has the same number of elements as the original array, and the values of the original array are copied into the elements of the array being created:

ch4/SimpleFloatArray.C

```
SimpleFloatArray::SimpleFloatArray(const SimpleFloatArray& a) {
    num_elts = a.num_elts;
    ptr_to_data = new float[num_elts];
    copy(a);    // Copy a's elements
}

void SimpleFloatArray::copy(const SimpleFloatArray& a) {
    // Copy a's elements into the elements of *this
    float* p = ptr_to_data + num_elts;
    float* q = a.ptr_to_data + num_elts;
    while (p > ptr_to_data) *--p = *--q;
}
```

After setting the number of elements and allocating the built-in array, the elements of the original array are copied into the new array by calling the member function copy(). This member function is a *private member function*: It can be called only by other members of SimpleFloatArray.

When a SimpleFloatArray object goes out of scope, C++ automatically calls the destructor for that object. The destructor uses delete[ ] to deallocate the built-in array used to hold the elements of the SimpleFloatArray:

ch4/SimpleFloatArray.C

```
SimpleFloatArray::~SimpleFloatArray() {
    delete [ ] ptr_to_data;
}
```

This code is correct even for SimpleFloatArray objects with no elements because

- When invoked with a null pointer, delete and delete [ ] have no effect.

The subscripting member simply returns a reference to the appropriate element of the built-in array, and the numElts() member returns the number of elements:

ch4/SimpleFloatArray.C

```
float& SimpleFloatArray::operator[ ](int i) {
    return ptr_to_data[i];
}

int SimpleFloatArray::numElts() {
    return num_elts;
}
```

Array assignment is implemented by setting the size of the left operand to the size of the right operand and copying the elements:

ch4/SimpleFloatArray.C

```
SimpleFloatArray& SimpleFloatArray::operator=(const SimpleFloatArray& rhs) {
    if ( ptr_to_data != rhs.ptr_to_data ) {
        setSize( rhs.num_elts );
        copy(rhs);
    }
    return *this;
}
```

The test handles the case in which an array is assigned to itself. Without the test, setSize() might delete the elements of the left operand, which would also be the elements of the right operand, before the values are copied.

- Assignment member functions should work correctly when the left and right operands are the same object.

The size of a SimpleFloatArray is set by deleting its elements, saving the new size, and allocating a built-in array with the appropriate number of elements; the old element values are not saved.

ch4/SimpleFloatArray.C

```
void SimpleFloatArray::setSize(int n) {
    if (n != num_elts) {
        delete [ ] ptr_to_data;          // Delete old elements,
        num_elts = n;                    // set new count,
        ptr_to_data = new float[n];      // and allocate new elements
    }
}
```

Finally, scalar assignment is implemented by a loop over the elements:

ch4/SimpleFloatArray.C

```
SimpleFloatArray& SimpleFloatArray::operator=(float rhs) {
    float* p = ptr_to_data + num_elts;
    while (p > ptr_to_data) *--p = rhs;
    return *this;
}
```

The member functions for our array class were selected by programmer design: C++ offers tools but does not dictate how they are used. Moreover, there are no dictates on the contents of member functions other than consistency with the function argument types and return type. While this freedom allows great range in possible designs, the programmer must learn typical patterns of usage to produce classes that are reliable and used easily.

One of those patterns of member functions applies to any class that allocates dynamic memory for each object. To ensure that copies of such objects both allocate appropriate new memory and delete memory allocated for each object, all constructors must allocate memory or set any memory pointers to zero, and the class needs a copy constructor, a copy assignment operator, and a destructor.

Chapter 7 explores the memory management issues in more detail. In practice, the coordination of member functions within a class can be ignored even with dynamic memory allocation simply by replacing built-in pointers with either programmer-defined pointers (see Chapter 14) or programmer-defined arrays (see Chapter 13), depending on whether single objects or arrays of objects are needed as member data. Of course these programmer-defined classes require coordinated member functions, as we described for the programmer-defined array class SimpleFloatArray.

The freedom of definition in member functions does not mean that programmer-defined classes in C++ are without restrictions. The language applies the same precedence and associativity to programmer-defined operators that it uses for the equivalent built-in operators (cf. Section 2.5) and the language requires that the same copy constructor be used for explicit initialization of objects, initialization of function arguments, and function returns. For these reasons, C++ classes that act like built-in types will be most successful. Thus we adopted a form for our SimpleFloatArray that parallels the use of C++ built-in arrays.

4.3 Class Templates

Classes provide a mechanism for defining new types. Often, however, one wants to define a family of closely related types, much like defining single precision, double precision, complex, and double precision complex versions of a FORTRAN subroutine. Usually the code for a particular subroutine is precision independent except for variable types and perhaps a few constants. A naming convention is used to distinguish among the various versions of a subroutine. A similar need arises when defining C++ classes.

For example, the SimpleFloatArray of Section 4.2 is written for float elements, but SimpleDoubleArray, SimpleIntArray, and so forth would be equally useful. Moreover, all of the classes would share an almost identical implementation: Change float to double (or whatever) and you have the new class.

A C++ *class template* expresses this kind of similarity. A class template is exactly what its name implies: a template for producing classes. Syntactically, a class template declaration consists of the keyword template, followed by a list of *template arguments* enclosed in angle brackets (< >), followed by a class declaration:

template<*template-argument*, ...> class *identifier*;

Each template argument is either the keyword class followed by an identifier, or an argument declaration as would appear in a function prototype. An array class template declaration might look like this:

ch4/SimpleArray.h

```
template<class T> class SimpleArray;
```

This says that SimpleArray is a *parameterized type* with T as a parameter; T must be a type. Specific classes produced by a class template declaration are denoted by the class template name followed by parameter values in angle brackets, like this: SimpleArray<int> or SimpleArray<float>. These types would correspond to arrays with int elements or float elements, respectively. A class produced from a class template is called a *template class* and a template class is said to be an *instance* of a class template. SimpleArray is a class template; SimpleArray<int> and SimpleArray<float> are template classes.

A class template definition has a class body:

```
template<template-argument, ...> class identifier {
    // ... class body ...
};
```

Within the class body, the template arguments declared with the class keyword can appear as type names; template arguments can also be compile-time constants like integers. Here is a class template for the parameterized array type declared earlier:

ch4/SimpleArray.h

```
template<class T>
class SimpleArray {
public:
    SimpleArray(int n);                                  // Create array of n elements
    SimpleArray();                                       // Create array of 0 elements
    SimpleArray(const SimpleArray<T>&);                  // Copy array
    ~SimpleArray();                                      // Destroy array
    T& operator[](int i);                                // Subscripting
    int numElts();                                       // Number of elements
    SimpleArray<T>& operator=(const SimpleArray<T>&);    // Array assignment
    SimpleArray<T>& operator=(T);                        // Scalar assignment
    void setSize(int n);                                 // Change size
```

```
private:
    int num_elts;                          // Number of elements
    T* ptr_to_data;                        // Pointer to built-in array of elements
    void copy(const SimpleArray<T>& a);    // Copy in elements of a
};
```

Compare this class template with the definition of SimpleFloatArray on page 97. Other than changing the class's name, we have simply replaced float with the type parameter T. Note that the names of constructor and destructor member functions do not include the parameter T. With this class template, the line-fitting program of Section 4.2 could be written (yet again) like this:

ch4/linefit2.C

```
#include <iostream.h>
#include "ch4/SimpleArray.h"

void linefit() {
    // Create arrays with the desired number of elements
    int n;
    cin >> n;
    SimpleArray<float> x(n);
    SimpleArray<float> y(n);

    // Read the data points
    for (int i = 0; i < n; i++) {
        cin >> x[i] >> y[i];
    }

    // ... same as before ...
}
```

Member function definitions for class templates are preceded by the template keyword with parameters in angle brackets, but are otherwise analogous to ordinary member function definitions. Here are the definitions forSimpleArray<T>:

ch4/SimpleArray.c

```
template<class T>
SimpleArray<T>::SimpleArray(int n) {
    num_elts = n;
    ptr_to_data = new T[n];
}

template<class T>
SimpleArray<T>::SimpleArray() {
    num_elts = 0;
    ptr_to_data = 0;
}
```

```
template<class T>
SimpleArray<T>::SimpleArray(const SimpleArray<T>& a) {
    num_elts = a.num_elts;
    ptr_to_data = new T[num_elts];
    copy(a);    // Copy a's elements
}


template<class T>
void SimpleArray<T>::copy(const SimpleArray<T>& a) {
    // Copy a's elements into the elements of *this
    T* p = ptr_to_data + num_elts;
    T* q = a.ptr_to_data + num_elts;
    while (p > ptr_to_data) *--p = *--q;
}


template<class T>
SimpleArray<T>::~SimpleArray() {
    delete [] ptr_to_data;
}


template<class T>
T& SimpleArray<T>::operator[](int i) {
    return ptr_to_data[i];
}


template<class T>
int SimpleArray<T>::numElts() {
    return num_elts;
}


template<class T>
SimpleArray<T>& SimpleArray<T>::operator=(const SimpleArray<T>& rhs) {
    if ( ptr_to_data != rhs.ptr_to_data ) {
        setSize( rhs.num_elts );
        copy(rhs);
    }
    return *this;
}
```

```
template<class T>
void SimpleArray<T>::setSize(int n) {
    if (n != num_elts) {
        delete [] ptr_to_data;              // Delete old elements,
        num_elts = n;                       // set new count,
        ptr_to_data = new T[n];             // and allocate new elements
    }
}
```

```
template<class T>
SimpleArray<T>& SimpleArray<T>::operator=(T rhs) {
    T* p = ptr_to_data + num_elts;
    while (p > ptr_to_data) *--p = rhs;
    return *this;
}
```

Again compare these with the member function definitions for SimpleFloatArray on page 97.

4.4 Function Templates

In the same way that it is useful to define parameterized types, it is useful to define parameterized functions in which the function body can be parameterized by argument types. For example, since C++ lacks a built-in exponentiation operator, we find it useful to define a sqr() function that squares any number. We wrote such a function in Sections 2.16.3 and 3.10.3 that takes a double and returns a double. We want a different function for each numeric type, each with essentially the same body. A *function template* can express this kind of similarity.

Syntactically, a function template declaration consists of the keyword template, followed by a list of template arguments enclosed in angle brackets (< >), followed by a function declaration:

template<*template-argument*, ...> *function-declaration*;

Each template argument must consist of the keyword class followed by an identifier, and each identifier must appear in the argument list of the *function-declaration*. If the *function-declaration* is also a definition, then the function template declaration is also a function template definition.

Here is a function template definition for sqr():

SciEng/utils

```
template<class T>
inline
T sqr(T x) {
    return x * x;
}
```

This says that sqr() is parameterized by the parameter T: A function is generated from the function template by substituting a specific type for the parameter T. The specific type is determined by the type of the argument passed to sqr(); that is why every template argument must appear in the function's argument list. Here are some examples:

ch4/sqr.C

```
int i = 1;
float f = 3.1;
double d = 4.4;
complex c(3, 4);
cout << sqr(i) << endl;    // Generate sqr(int)
cout << sqr(f) << endl;    // Generate sqr(float)
cout << sqr(d) << endl;    // Generate sqr(double)
cout << sqr(c) << endl;    // Generate sqr(complex)
```

The detailed rules for determining when a function template is expanded into a function are somewhat complicated, and we shall defer discussing them until Section 5.6. For now, it is sufficient simply to understand that function templates exist and behave as one would intuitively expect for simple cases.

4.5 Exceptions

Programs must handle errors. The easiest approach is to print an error message and terminate the program. This approach is unthinkable in an interactive program, where the user expects to be able to make a mistake, get a helpful message, and try again. Such programs are themselves built from classes and functions: There must be agreement among the authors of classes and of functions using the classes about how to handle errors. A member function that terminates the program (by calling the exit() function declared in the standard stdlib.h header file) when it detects an error can't be used in an interactive program. (Strictly speaking, this is not true: The calling program could check the arguments to the member function to make sure that no error will arise. However, doing so can require almost as much work as writing the member function itself.) If the author of each class adopts a different error-reporting convention (setting a status argument, setting a global variable, calling an error subroutine, etc.), it becomes difficult to build a program using classes from different sources. Therefore C++ provides a mechanism, called *exceptions*, for reporting and handling errors. (Exceptions are a relatively new feature of C++; see Notes and Comments 4.4.)

The idea is simple: The function that detects an error *throws* an object, which can be caught by code that called the function (directly or indirectly). We illustrate this in the context of a simple interactive program:

ch4/interactive.C

```
#include <iostream.h>
#include <string.h>

extern void command_a();
extern void command_b();
extern void command_c();

int main() {
    char command[100];
    while (cin >> command) {
        if      (strcmp(command, "a") == 0) command_a();
        else if (strcmp(command, "b") == 0) command_b();
        else if (strcmp(command, "c") == 0) command_c();
        // ...
        else    cout << "Unknown command: " << command << endl;
    }
    return 0;
}
```

The function strcmp() is declared in the standard header string.h and returns 0 if its two char* arguments compare equal. The program loops, reading commands from the standard input. If the command is recognized, a function is called to perform the command.

Now suppose that the command functions, or functions called by them, can detect incorrect input. When incorrect input is detected, they throw an exception:

ch4/interactive2.C

```
void command_a() {
    cout << "Doing a" << endl;
    float x;
    cin >> x;
    if (x < 0) {
        throw "Nonnegative input expected in command a";
    }
    // ... process the command with valid input ...
}
```

An exception is thrown by the throw statement, which throws an object, here a char*. When a function executes a throw statement, the function's execution is terminated and is not resumed. In this example, the code for processing the command with valid input is not executed if a negative number is read from the input.

Code enclosed in a try-block expects to handle exceptions; exceptions are caught by an *exception handler*, which appears immediately following the try-block. Here is the syntax:

```
try {
    ...
} catch (argument-list) {
    statement;    // exception handler body
    ...
}
...
```

There can be one or more catch clauses associated with a try-block.

Here is the main loop of the preceding interactive program, modified to catch exceptions:

ch4/interactive2.C

```
int main() {
    char command[100];
    while (cin >> command) {
        try {
            if      (strcmp(command, "a") == 0) command_a();
            else if (strcmp(command, "b") == 0) command_b();
            else if (strcmp(command, "c") == 0) command_c();
            // ...
            else    cout << "Unknown command: " << command << endl;
        }
        catch (char* message) {
            cout << message << endl;
        }
    }
    return 0;
}
```

The calls to the command-processing functions are enclosed in a try-block. If an exception is thrown while the try-block is active and the thrown object is a char*, then message is initialized from the thrown object (like function argument initialization) and the body of the exception handler is executed. Execution does not return to the function that threw the object.

More generally, any kind of object can be thrown or caught as an exception, with different kinds of objects used in different situations. Consider a new version of SimpleArray, called CheckedSimpleArray, that checks both subscript range and the number of elements requested when creating the array. Two different types can be used to distinguish between these errors:

ch4/CheckedSimpleArray.h

```
class SubscriptRangeError {
public:
    SubscriptRangeError(int i);
    int badSubscript();
private:
    int subscript;
};

class ArraySizeError {
public:
    ArraySizeError(int n);
    int badSize();
private:
    int size;
};
```

The CheckedSimpleArray class definition is the same as the definition of SimpleArray except for changing the name. Three member functions must be modified to throw exceptions:

ch4/CheckedSimpleArray.c

```
template<class T>
CheckedSimpleArray<T>::CheckedSimpleArray(int n) {
    if (n < 0) throw ArraySizeError(n);
    num_elts = n;
    ptr_to_data = new T[n];
}

template<class T>
T& CheckedSimpleArray<T>::operator[](int i) {
    if (i < 0 || i >= num_elts) throw SubscriptRangeError(i);
    return ptr_to_data[i];
}

template<class T>
void CheckedSimpleArray<T>::setSize(int n) {
    if (n != num_elts) {
        if (n < 0) throw ArraySizeError(n);
        delete [] ptr_to_data;                  // Delete old elements,
        num_elts = n;                           // set new count,
        ptr_to_data = new T[n];                 // and allocate new elements
    }
}
```

These exceptions can be caught as illustrated in this program fragment:

ch4/tCheckedSimpleArray.C

```
try {
    CheckedSimpleArray<float> a1(-3);    // Bad size
}
catch(ArraySizeError e) {
    cerr << "ArraySizeError caught, bad size = " << e.badSize() << endl;
}
try {
    CheckedSimpleArray<float> a(3);
    a[3];   // Bad subscript
}
catch(SubscriptRangeError e) {
    cerr << "SubscriptRangeError caught bad subscript = " << e.badSubscript() << endl;
}
```

After one of the catch clauses associated with a try-block is executed, program execution continues immediately following the last catch clause.

Of course, these fragments are contrived because the try-blocks only contain the code that causes the exceptions to be thrown. Here is a more typical example:

ch4/tCheckedSimpleArray.C

```
try {
    int n;
    cin >> n;
    CheckedSimpleArray<float> a(n);

    // ...

    int j;
    cin >> j;
    float x = 2.1 * a[j];

    // ...
}
catch(SubscriptRangeError e) {
    cerr << "SubscriptRangeError caught bad subscript = " << e.badSubscript() << endl;
}
catch(ArraySizeError e) {
    cerr << "ArraySizeError caught, bad size = " << e.badSize() << endl;
}
```

If the value of n is bad, the CheckedSimpleArray constructor will throw ArraySizeError; the remainder of the try-block will be skipped and the catch clause taking the type

of the thrown object (ArraySizeError) is executed. If creation of the array succeeds, a value for j will be read and used in the expression a[j]; if the value is bad, the subscript operator member function will throw SubscriptRangeError; the remainder of the try-block will be skipped and the catch clause taking an ArraySizeError will be executed. The important point is that the first catch clause that matches the type of the thrown object is executed.

If an exception is thrown that is not caught by an exception handler, the program terminates. Thus in their simplest form, exceptions provide a uniform mechanism for terminating program execution when an error is found; where more sophisticated error handling is useful, exceptions can be caught and handled appropriately. We will see such examples throughout this book.

4.6 Nested Classes

A *nested class* is a class that is defined within the definition of another class, as illustrated by the following code:

ch4/Nested.C

```
class Outer {
public:
    // ...
    class Inner1 {
        // Class body for Inner1
    };
    // ...
private:
    // ...
    class Inner2 {
        // Class body for Inner2
    };
    // ...
};
```

Classes Inner1 and Inner2 are nested in class Outer. Except for its name, a nested class is the same as any other class. Within the enclosing class, a nested class name must be unique and can be referred to using just its name; from outside the enclosing class, a nested class must be named with the scope resolution operator :: (e.g., Outer::Inner1) example. Nested class names obey the usual access rules: Outer::Inner2 is private and can't be accessed outside of Outer.

Nested classes can be used to group related classes. Consider a class QuadraticPolynomial that holds a quadratic polynomial $ax^2 + bx + c$. It might have member functions realRoots() to obtain the real roots, complexRoots() to obtain the complex roots, and evaluateAt() to evaluate the polynomial at a specified x. How should the roots be returned? The root-finding functions could set two arguments passed by

reference; however, the roots are a pair of related numbers, so we choose to define two root classes, RealRoots and ComplexRoots. What should happen if you call realRoots() and the polynomial only has complex roots? We choose to have realRoots() throw an instance of a NoRealRoots class.

With these decisions made, we could simply write four independent classes. But doing so would lose the relationship among these four classes. Moreover, if we were to write a CubicPolynomial class, we might wish we had chosen names like QuadraticRealRoots and QuadraticComplexRoots. Nested classes solve both problems. We write QuadraticPolynomial enclosing the other three classes:

ch4/quadratic.C

```
class QuadraticPolynomial {
public:
    QuadraticPolynomial(double a, double b, double c);

    class RealRoots {
    public:
        RealRoots(double x1, double x2);
        double minRoot();
        double maxRoot();
    private:
        double min_root;
        double max_root;
};

class ComplexRoots {
    // ...
};

RealRoots    realRoots();              // Solve for real roots
ComplexRoots complexRoots();           // Solve for complex roots
double       evaluateAt(double x);     // Evaluate at x

    class NoRealRoots {
        // Exception: real roots requested but both roots are complex.
        // ...
    };
private:
    // Polynomial coefficients
    double a;
    double b;
    double c;
};
```

Within the enclosing class, we can write RealRoots, ComplexRoots, and NoRealRoots; outside the enclosing class, we must use the scope resolution operator :: to *qualify* those names, like this:

ch4/quadratic.C

```
// ... Set a, b, c ...
try {
    QuadraticPolynomial eqn(a, b, c);
    QuadraticPolynomial::RealRoots roots = eqn.realRoots();
    cout << "Real roots are: " << roots.minRoot() << ", " << roots.maxRoot() << endl;
    // ... computation using the real roots ...
}
catch (QuadraticPolynomial::NoRealRoots) {
    cout << "No real roots." << endl;
}
```

Our intent is clear, and there is no potential name conflict with CubicPolynomial.

The scope resolution operator must be used when defining member functions of an enclosing class that return nested class objects or when defining member functions of nested classes. As an example of the former, here is the definition of realRoots():

ch4/quadratic.C

```
QuadraticPolynomial::RealRoots QuadraticPolynomial::realRoots() {
    // Solve for real roots.
    // Formula from Section 5.6, Numerical Recipes in C, 2nd ed.,
    // by Press, Teukolsky, Vetterling, and Flannery,
    // Cambridge University Press, 1992.
    double sgn_b = (b >= 0.0 ? 1.0 : -1.0);
    double disc = b*b - 4*a*c;
    if (disc < 0.0) throw NoRealRoots();
    double q = -.5 * ( b + sgn_b * sqrt(disc) );
    return RealRoots( q/a, c/q );
}
```

We had to qualify the return type in the function definition because the type name appears outside of the enclosing class. However, once the function identifier QuadraticPolynomial::realRoots() has been seen, we are inside the enclosing class scope and there is no need to qualify the nested class names. Thus in the function body we write NoRealRoots and RealRoots, without qualification; although not necessary, qualification is permitted.

To illustrate the syntax for defining members of nested classes, here are the definitions for the members of RealRoots:

ch4/quadratic.C

```
QuadraticPolynomial::RealRoots::RealRoots(double x1, double x2) {
    if (x1 < x2) {
        min_root = x1;
        max_root = x2;
    }
    else {
        min_root = x2;
        max_root = x1;
    }
}
double QuadraticPolynomial::RealRoots::minRoot() { return min_root; }
double QuadraticPolynomial::RealRoots::maxRoot() { return max_root; }
```

Outside of the enclosing class, the name QuadraticPolynomial::RealRoots must be used to refer to the nested RealRoots class. Thus that name prefixes the member function names in the member function definitions.

We use nested classes throughout this book.

- Use class nesting to group related classes that work together.

We shall also see that class nesting can be a powerful tool when writing class templates.

4.7 Overview of C++ Programs

With some understanding of classes, we are now ready to give you the big picture: how the elements of C++ are combined to create C++ programs. At the same time, we can relate our discussion of basics to the more advanced material to come.

Structurally, a typical C++ program appears as in Figure 4.1. A C++ *program* consists of one or more files that are linked together. One file contains a special function named main(), which is called by the operating system to start the program. The main() function creates objects from classes, which are declared and defined in header files included in the file containing main(). The creation of these objects means constructor member functions will be called, and this may cause other objects to be created and functions to be called. The main() function then calls member functions for the objects to accomplish the work of the program; the member functions may create other objects or call other functions. The member functions are declared in class definitions contained in the class's header file; they are defined in the class's implementation file. Within each function call, local objects are destroyed when the function completes; when main() completes, the objects it has created are also destroyed.

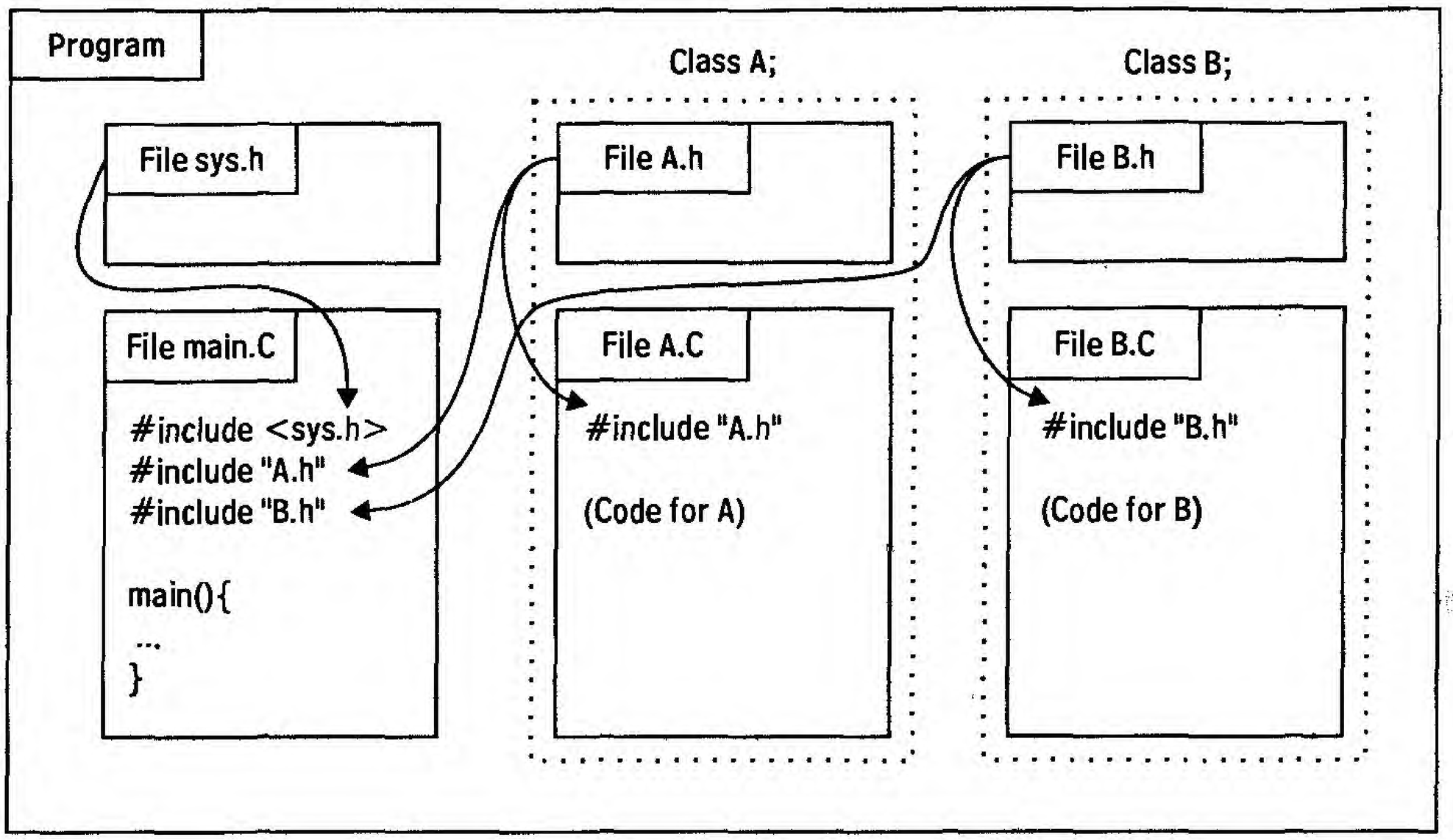


Figure 4.1 The Big Picture. A typical C++ program consists of one main() implementation file (called main.C here) linked with implementation files for classes (files A.C and B.C in this example). Header files containing the class definitions (files A.h and B.h) connect the use of the classes to their implementations. Nonclass functions must also be declared in header files (as in sys.h here); their implementations often come from system libraries.

C++ programs also use nonmember functions, especially non-C++ functions from the operating system or non-C++ libraries. These are declared via extern function prototypes in separate header files. All of the function calls, whether member functions or not, are checked both at definition and use for correct argument types.

The critical separation of the main() function from the implementation of the classes allows the main() function to look at the classes only through the header files, and for one class to look at other classes only via the declarations in the header files. By combining data with functions to create objects, C++ generalizes the concept of subroutine libraries. (This idea is explored in Chapter 15.) Every C++ program should build on other classes, and—after an initial phase of class construction—portions of most C++ programs should be built from existing components.

4.8 Notes and Comments

4.1 The distinction between a declaration and a definition in C++ is muddled by the terminology often used in the C++ literature. At the risk of making matters worse we have adopted a more uniform, but overlapping terminology. Fundamentally, a decla-

ration introduces a name into a program [44, Section 3.1]. A definition is a declaration that also associates something with the name. Every definition is a declaration, but not vice versa. There can be many declarations of a name, but only one definition.

A function prototype is a declaration only; a function with a (possibly empty) body enclosed in braces is both a declaration and a definition. We call these a function declaration and a function definition, respectively; so does the C++ literature. The keyword class followed by a name declares that name to be a type. We call this a class declaration; there is no commonly used name for this in the literature. The keyword class followed by a name and a (possibly) empty class body within braces associates data and function members with a name. We therefore call it a class definition; for historical reasons [44, Section 9] this is called a class declaration in the literature.

4.2 A good technique in FORTRAN programming is to place a group of related variables in a COMMON block and write subroutines to manipulate the variables. Some of the subroutines will load data into the COMMON block, others will access it, while still others might transform it in some way. Not only does this technique reduce the overhead of passing arguments in subroutine calls, but it is a good tool for organizing complex FORTRAN code. This organization is akin to a simplistic view of a C++ class, with the COMMON block analogous to a class instance and the subroutines analogous to the member functions. Of course, in the FORTRAN scheme only one instance can be grouped with the subroutines at a time. The C++ class serves as a stamp for creating an unlimited number of instances with a single set of (member) functions.

4.3 The means provided for manipulating many identically formatted data items (e.g., Points or Lines) are vital distinctions among programming languages. Even modest-sized FORTRAN programs often contain hundreds of arrays, groups of which are meant to be indexed together to obtain a representation of a group of objects. For example, 100 Line objects might be represented by

ch4/linest.f

```
REAL LINEA, LINEB, LINEC
COMMON /LINEST/LINEA(100), LINEB(100), LINEC(100)
```

C is more flexible in allowing related items to be grouped:

ch4/cLine.C

```
struct Line { float a; float b; float c; };
struct Line bar[100];
```

However, lacking a concept like member functions, C encourages the manipulation of these structs via pointers, and soon the nature of the manipulations becomes obscured in pointer notation. C++ retains C's structs, extending them slightly by allowing them to have member functions. C++ classes subsume structs, and structs are rarely used.

4.4 Templates and exceptions are recent additions to C++. They are described in [44] as experimental features of the language. Since publication of [44], both have been adopted in the ANSI draft standard, albeit in somewhat modified form. As of early 1994, several compilers that implement both templates and exceptions are commercially available, and many commercially available compilers implement templates.

There is, however, substantial variation in the mechanisms used to create template classes from class templates and template functions from function templates. Some compilers determine automatically which classes and functions to generate; with other compilers, the user must list them. The different mechanisms used lead to differences across compilers in the way header and source files should be structured. Refer to the manual for your compiler.

Exceptions and, especially, templates are powerful programming tools; we use both. In fact, we deem templates to be essential for scientific and engineering programming.

4.9 Exercises

4.1 The following code compiles and runs but produces wrong answers.

ch4/distance2.C

```
#include <iostream.h>
#include "ch4/Point.h"

int main() {
    // Read coordinates of a point and compute its distance to the origin
    Point origin;
    float x; cin >> x;
    float y; cin >> y;
    Point p(x, y);
    cout << origin.distance(p) << endl;
    return 0;
}
```

Why? Modify the Point class so that it can detect these kinds of errors.

4.2 Add a moveTo(Number, Number) member function to Point.

4.3 Add to the Point class a member function called angle() that takes two Point arguments p_1 and p_3 and returns the positive angle $\angle p_1 p_2 p_3$, where p_2 is the Point object with which angle() is called. *Hint:* Let v_{ij} be the vector from p_i to p_j and let θ be the desired angle. From elementary analytic geometry, $v_{21} \cdot v_{23} = |v_{21}||v_{23}| \cos\theta$ and $|v_{21} \times v_{23}| = |v_{21}||v_{23}| \sin\theta$. Therefore, $\theta = \tan^{-1}(|v_{21} \times v_{23}|/(v_{21} \cdot v_{23}))$.

4.4 Modify the Point class to store polar coordinates. What additional member functions would you add? For which member functions would revised definitions be needed?

4.5 Redesign the Point and Line examples in this chapter for three dimensions.

4.6 If you designed a C++ program to compute the force between electrostatically charged particles according to the formula

$$f = \sum_i \sum_{j \neq i} kq_i q_j / |\mathbf{r}_i - \mathbf{r}_j| \tag{4.1}$$

where k is a constant giving the units, q_i is the charge on particle i, and $\mathbf{r}_i$ is the position of particle i, what kinds of objects would you select? What member functions would you choose? Complete the implementation.

4.7 Review the results of the previous exercise: Did you use the coulombsLaw() function from Chapter 2 and the three-dimensional Point class from Exercise 4.5? Why or why not?

4.8 Our version of intersect() for Line on page 94 uses parallelism_tolerance defined to be the angle θ (in radians) between the lines. Change the program to use $\sin^2\theta$ for the tolerance, to avoid recomputing the test for each call to intersect(). Notice that you must change both the code inside Line, the function intersect(), and the code outside of Line any time you reset the tolerance. Change the original program again; this time make parallelism_tolerance a private static variable and add a static member function, set_parallelism(), to set the value. Redo the first part of this exercise with the new version. Notice that the internal interpretation can now be changed without altering any code that resets the tolerance because set_parallelism() hides the new interpretation.

4.9 For each statement in the original line-fitting program (starting on page 48) that uses an array, identify the member function of SimpleFloatArray that does the equivalent pointer manipulation.

4.10 Template arguments for class templates need not be types. For example, the declaration

ch4/FixedArray

```
template<class T, int n> class FixedArray;
```

declares a class template with two template arguments, the second an int. The actual value of a template argument must be constant at compile time. Thus FixedArray<int, 4> is legal, but FixedArray<int, i>, with i an int variable, is not. Design, implement, and test a FixedArray<T, n> class that provides arrays of n elements of type T. Discuss the advantages and disadvantages of this class compared with SimpleArray<T>.

4.11 Parameterized types can be used as arguments to templates. Using the FixedArray<T, n> class template from the preceding exercise, write a variable declaration for a 3×4 array of int elements. How would you write an expression to access the (2, 3) element of the array? *Hint:* Do not write a new class.

4.12 Write a version of SimpleArray<T> that allows the indexing origin to be set at compile time. Write another version that allows the indexing origin to be set at runtime.

4.13 Our implementation of the intersect() member function of the Line returns a point-at-infinity result. Modify intersect() to throw an exception when there is no reasonable point of intersection.

4.14 Compare the advantages and disadvantages of using exceptions to report error conditions versus passing to functions an extra integer argument that is set by the function to indicate status.

CHAPTER 5

Functions

We skimmed the basics of C++ functions in Sections 2.16 and 3.10. This chapter digs deeper into functions, to build a foundation for discussing, in Chapter 6, the role of functions in classes.

A function specifies a sequence of operations, packaged together, that may be called from different parts of a program. Matching a function call to the corresponding definition requires a contract between the caller and called function. In C++, this contract comes in the form of a function declaration, the first concept we take up in this chapter.

5.1 Declarations and Definitions

Practical matters dictate that programs larger than a few hundred lines of code be split into pieces that can be written, compiled, and tested separately. Each portion of a C++ program to be compiled separately is called a *translation unit;* on most systems, each translation unit is compiled into an object file, sometimes called a "dot oh" (".o") file. Without safeguards, compiling translation units separately could lead to type errors: When code in one translation unit calls a function in another translation unit, C++ must ensure that the proper number of arguments of the proper types are passed and that the proper type of object (if any) is returned.

C++ therefore requires that

- A function must be declared before it can be called.

A *declaration* is a specification, a promise that an identifier will have certain properties; a *definition* is an implementation of the specification that fulfills the promise. The declaration/definition relation for function identifiers parallels that of objects and classes. For example,

ch5/dcldef.C

```
extern int day_of_year;
extern double f(double);
```

are both declarations. The first declaration says that the name day_of_year refers to an int value that is defined elsewhere; the second declaration says that the name f refers to a function that takes a single argument of type double and returns a value of type double. The first declaration does not give day_of_year a value; the second declaration does not say what the function does. These declarations promise that day_of_year has type int and that there is a function with the name f that takes a double and returns a double. Corresponding definitions might look like these:

ch5/dcldef.C

```
int day_of_year = 31;
double f(double x) {
    return 1.0 / x;
}
```

They each define an entity that fulfills the promise made in the corresponding declaration. Technically, all definitions are also declarations; see Notes and Comments 5.3.

The distinction between declaration and definition reflects the way we use abstraction in our thinking. For example, we are comfortable saying that there is a sin function without knowing its implementation. The declaration

ch5/dcldef.C

```
extern double sin(double);
```

says that there is a sin function that takes a double argument and returns a double result. A *function declaration* specifies the function's identifier (name), argument types, and return value type. The compiler checks that every function call is consistent with the specification given by a function declaration.

Consistency across translation units is achieved by sharing a single copy of each function declaration among all translation units that use the function. Declarations to be shared among translation units are put into *header files*, usually grouped according to purpose; definitions are put into one or more *implementation files*, again usually grouped by purpose. Each implementation file has an #include command for each header file that contains a declaration needed by code in the implementation file. A part of the C++ compiler called the C++ *preprocessor* creates a complete translation unit by combining the specified header files with the implementation file, as illustrated in Fig. 5.1. This scheme ensures that the declarations are consistent across all translation units, and the compiler is able to check all function calls against the single copy of the function declaration.

Since a function declaration is only a specification, it is harmless to declare the same function more than once. Functions that are not called need not be defined. However, every function that is called must be defined exactly once.

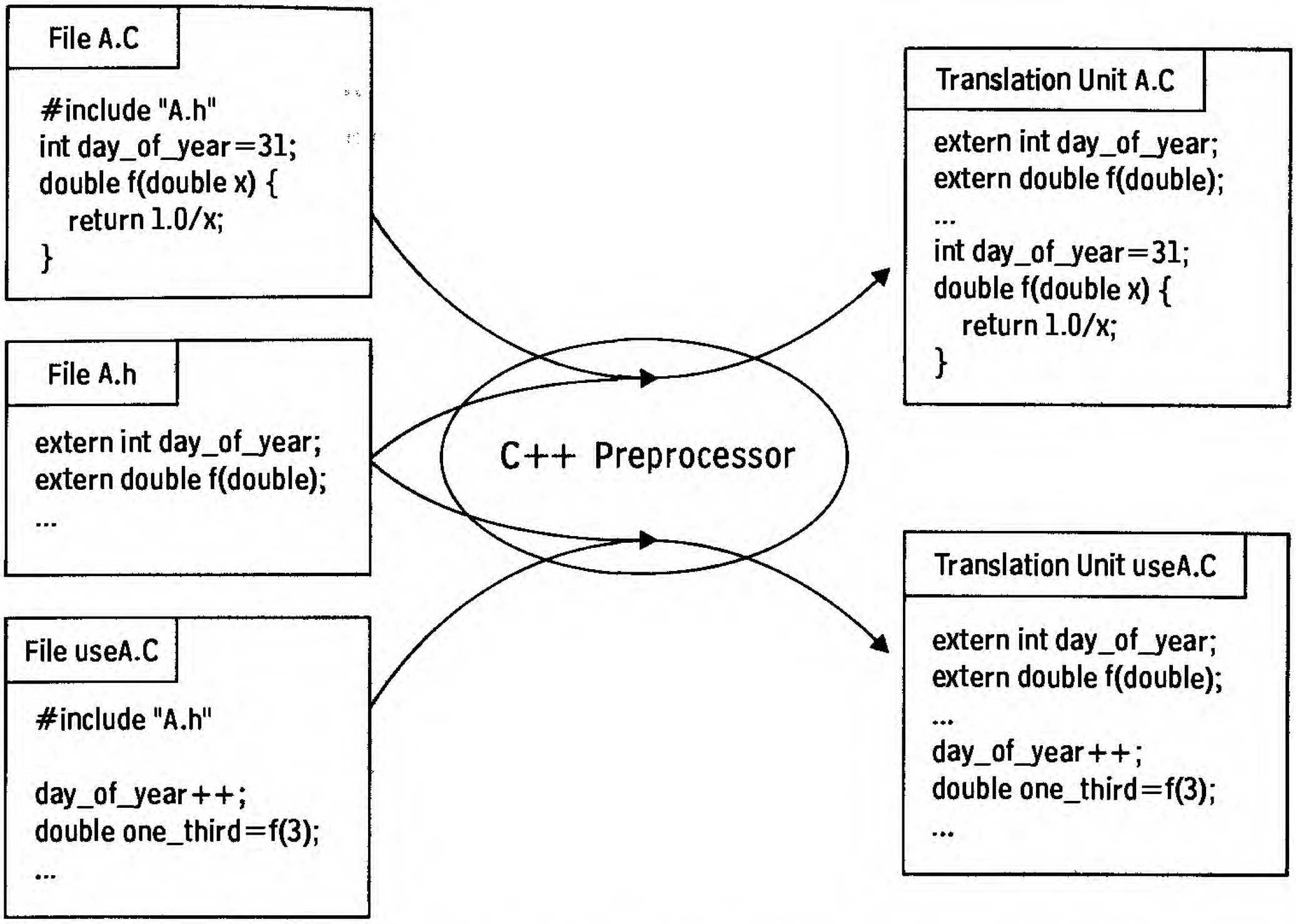


Figure 5.1 Use of Declarations in Header Files to Ensure Consistency of Declarations across Translation Units. Two declarations are grouped into the header file A.h. As the C++ preprocessor creates a translation unit from each of the implementation files, it replaces the #include with the contents of the header file A.h. Thus both translation units share consistent declarations, and the compiler can check that the definitions in file A.C and the uses of day_of_year and f() in file useA.C are consistent with the declarations.

5.2 Function Declarations

A function declaration consists of up to three components: an optional *linkage specifier*, a *function identifier*, and a *function type*. For example,

ch5/dcldef.C

```
extern double atan2(double x1, double x2);    // C++ function for arctan
```

has a linkage specifier of extern, a function identifier of atan2, and a function type of double()(double, double). The linkage specifier extern states that the name of this function, atan2, is not local to the translation unit it is declared in. In other words, all translation units that declare this function refer to the same function.

The C++ function atan2() shares some features with the mathematical function $\tan^{-1}$ and has several differences. We shall see examples of each as we describe the function declaration components in more detail.

5.2.1 Type

The function type determines the applicability of the function to a given set of arguments. The function type further breaks down into a *return type* and an *argument declaration*, analogs of the mathematical concepts of range and domain, respectively. In the function type from the preceding example, double()(double, double), the first double is the return type, the empty parentheses are a place holder for a function identifier, and the argument declaration contains two arguments, each of type double. The return type specifies the range of the function in a mathematical sense, and the argument declaration specifies the acceptable inputs to the function, the analog of a mathematical function's domain. In mathematical notation, the function type for this example corresponds to $\Re \times \Re \rightarrow \Re$, for $\Re$ the set of reals.

5.2.2 Identifier

The function identifier is a name and obeys the usual rules for C++ identifiers. A C++ function identifier can refer to different functions if the arguments of these functions have different types, or if the functions are members of different classes. Consequently function identifiers are closer in meaning to function names as we use them in mathematics than they are to function names in FORTRAN and C. In mathematics, $\tan^{-1} : \Re \times \Re \rightarrow \Re$ denotes a function named $\tan^{-1}$ that maps values in the set $\Re \times \Re$ (ordered pairs of real numbers) to values in the set $\Re$ (real numbers). The C++ function identifier and function type correspond to the mathematical notation like this:

$$\underbrace{\tan^{-1}}_{\text{function identifier}} : \underbrace{\underbrace{\Re \times \Re}_{\text{argument declaration}} \rightarrow \underbrace{\Re}_{\text{return declaration}}}_{\text{function type}}$$

The function identifier, which is intended to indicate what the function does, provides information that is independent of the information provided by the function type: We can give a group of functions the same identifier to signal that they perform the same action on different data, or we can give a group of functions the same type to signal that they transform the same data in different ways. Thus in mathematics we use the identifier sin for both the real function $\sin : \Re \rightarrow \Re$ and the complex function $\sin : \mathcal{C} \rightarrow \mathcal{C}$. We mean for these functions to share certain properties, but they are not the same functions. Likewise,

ch5/dcldef.C

```
extern double  sin(double x);
extern complex sin(complex x);
```

are similar in our mind but have different definitions. Using the same name for multiple functions is called *overloading*, and the function name is said to be *overloaded*. Overloaded functions must differ in their argument types; C++ prohibits having multiple functions with the same argument types but different return types:

ch5/dcldef.C

```
extern double sin(double x);
extern float  sin(double x);  // WRONG: Return type differs from previous return type
```

The function identifier distinguishes a particular mapping from among all functions with the same function type, just as the mathematical function name distinguishes a particular mapping from among all mappings with the same domain and range. The C++ functions

ch5/dcldef.C

```
extern double cos(double x);
extern double sin(double x);
```

both transform double arguments to double return values, but the mappings are different.

5.2.3 Linkage

A function can have either *internal linkage*, callable only by functions defined in the same translation unit, or *external linkage*, callable by any function. A nonmember function is given internal linkage by using the static or the inline keyword in its declaration; otherwise nonmember functions have external linkage. For example, suppose several functions in a translation unit need an exponentially distributed random deviate with mean μ and that a function random() that returns uniform random deviates is available from a library. You could write the following code:

ch5/expdev.C

```
#include <math.h>
extern double random();   // Returns uniform deviates in (0, 1)

static double exp_random(double mu) {
    // Generates exponentially distributed random deviates using
    // the transformation method described in Knuth, 2nd ed.,
    // Seminumerical Algorithms, Section 3.4.D, Addison-Wesley, 1981.
    return -mu * log(random());
}

void simulation1() {
    // ...
    double x1 = exp_random(2.1);
    // ...
}
```

```
void simulation2() {
    // ...
    double x2 = exp_random(3.5);
    // ...
}
```

The static specifier on function exp_random() gives it internal linkage. It can't be called from outside of the translation unit in which it is defined. On the other hand, the functions simulation1() and simulation2() have external linkage, which means that they are callable from any translation unit.

Notice that we used the extern keyword in the declaration of random() but not in the declarations of simulation1() and simulation2(). All three functions have external linkage, but we suggest the following convention:

- When declaring a name with external linkage, omit the extern keyword in definitions and use the extern keyword in declarations.

For example,

ch5/dcldef.C

```
extern double x;
```

is a declaration of x, while

ch5/dcldef.C

```
double x;
```

is a definition. We adopt this simple convention to avoid contending with C++'s complicated rules for determining linkage.

Functions with internal linkage are used infrequently in programs that are written in an object-oriented style. They are useful when manipulating objects of built-in type, which have no member functions, or when using existing code written in C.

5.3 Function Arguments

Mathematical functions specify a mapping of their input value(s) to their output value. But in most programming languages, including FORTRAN, C, and C++, a function can also modify the value of various objects or even external device states. Anything that occurs during the execution of a function other than a mapping of the input to the output is called a *side-effect*. Examples of side-effects include modifying the value of actual arguments, altering the value of a variable in COMMON in FORTRAN, or altering a global variable in C.

Side-effects can be classified into *input modification side-effects* and *pure side-effects*. Functions with input modification side-effects change one or more argument values, while functions with pure side-effects alter some other memory locations or device states in a manner that survives completion of the function.

Side-effects signal trouble. A simple function with side-effects is not too hard to understand, but you quickly forget the ramifications of a few dozen or a few hundred functions with side-effects. Each function has an argument list (inputs) and a return value (output): You can see these and understand them. But to remember the function's side-effects usually means rereading the function. Even worse, your functions may be used by others who can't or won't read the functions to look for the side-effects.

Unfortunately, side-effects are vital to writing efficient programs in C++, FORTRAN, and most other languages. The key to resolving this conflict is to use side-effects in a disciplined way. Classes help make pure side-effects more predictable by grouping related data into objects with member functions to set, read, and manipulate that data.

- Pure side-effects are best provided in the context of classes.

Since class objects and member functions can express most computations more convienently and robustly than global functions,

- Avoid pure side-effects in global functions.

Controlling input-modification side-effects without sacrificing efficiency is the subject of the remainder of this section. The key issue is the relationship between a function's formal arguments and its actual arguments. This relationship can be controlled using the const, &, and * type modifiers. Overall, the remainder of this section elaborates on this message:

- Signal input-modification side-effects by consistent use of type modifiers in the declaration of the formal arguments.

5.3.1 Call by Value

Input-modification side-effects can be prevented by not allowing a function direct access to its actual arguments. Instead the actual arguments are used to initialize the function's formal arguments, and thereafter the function accesses the formal arguments. When a C++ function is called, a new object is made for every formal argument (except, as described later, reference arguments and arrays) and initialized from the corresponding actual argument.

Let's look at a simple example:

ch5/args.C

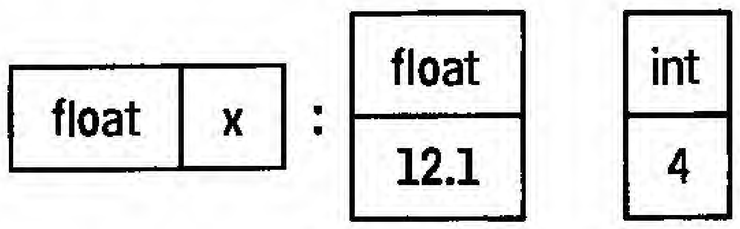
```
float s(float a, float b) {
    a = b;
    return a * b;
}
```

Consider what happens when

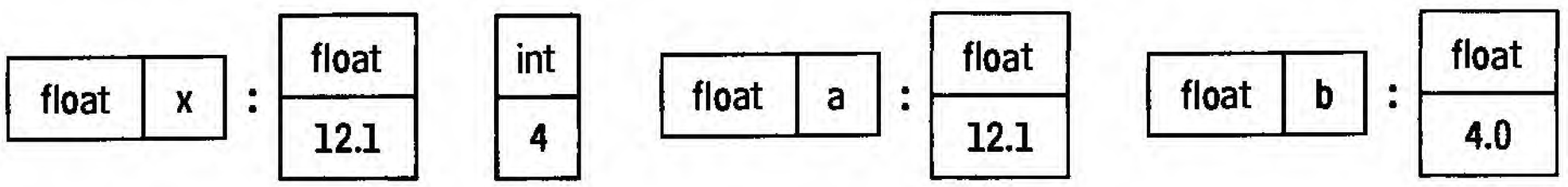
ch5/args.C

```
float x = 12.1;
s(x, 4);
```

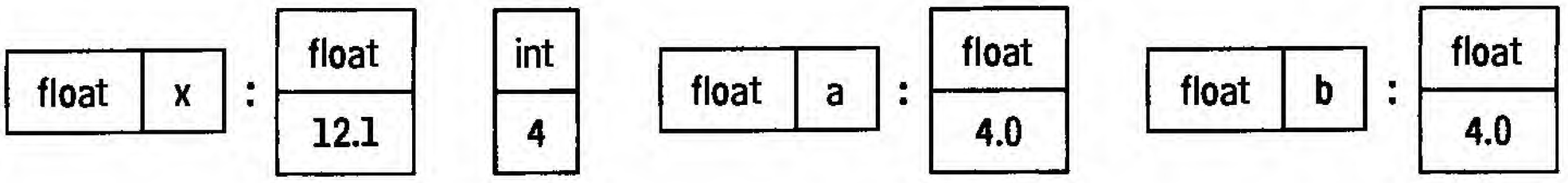
is executed. Before calling s, we have these objects:



The second actual argument is an object that is not associated with a variable. As s() is called, two new objects are created (one for each formal argument) and initialized from the actual arguments. The resulting objects are



Then, after the assignment statement is executed, we have



The object associated with x is not changed by the assignment in s; only the value of the object associated with a is changed. When s finishes, the new objects are discarded and x remains unaltered. This behavior is called *call by value*, a term that arises from the idea that the argument's value, but not the argument itself, is passed to a function; the arguments are said to be *passed by value*.

Initializing each formal argument from the corresponding actual argument can also cause type conversion. In the preceding example, the temporary float object created for the formal argument b has a different type than the actual int argument. To initialize the temporary object, a conversion from the type of the actual argument to the type of the formal argument was done, just as if the code

ch5/args.C

```
float b = 4;
```

had been executed. Consequently the usual arithmetic conversions (cf. page 23) are applied when arguments of built-in type are passed by value.

For class objects, initialization (and hence call by value) is done by invoking the copy constructor. When the types of the formal and actual arguments agree, the copy constructor will be called; Section 6.3 will discuss this. When the types

differ but the formal argument type has a single argument constructor matching the actual argument type, conversion construction is applied; Section 6.4.1 will discuss this. When the types differ but the actual argument type has a conversion operator matching the formal argument type, conversion will be applied; Section 6.4.2 will discuss this.

We will discuss copy constructors in detail in Section 6.3; for the moment it suffices to know that a copy constructor is a member function that takes a reference to an object and creates a copy of that object.

5.3.2 Call by Reference

Despite the simplicity and safety (through avoidance of side-effects) of call by value, it is inappropriate when an input-modification side-effect is desired or when copying a large object given as an actual argument is too expensive.

For input modification, a *reference argument*, indicated by an ampersand (&) following the type of the formal argument, can be used. For example, a function that orders its arguments can be written like this:

ch5/swap.C

```
int order(int& i1, int& i2, int& i3) {
    // Rearrange arguments in ascending order. Return maximum value.
    if (i1 > i2) { int temp = i1; i1 = i2; i2 = temp; }
    if (i2 > i3) { int temp = i2; i2 = i3; i3 = temp; }
    if (i1 > i2) { int temp = i1; i1 = i2; i2 = temp; }
    return i3;
}
```

and called like this:

ch5/swap.C

```
int a = 1;
int b = 2;
int c = -8;
order(a, b, c);
```

This usage is a particular case of the general notion of references to objects, described in Sections 2.15 and 3.9. A reference is another name for the object it is initialized with. Thus the formal arguments i1, i2, and i3 act like new names for the actual arguments passed to order(). No copying is done and modifying the formal arguments modifies the actual arguments. Using a reference formal argument converts the usual C++ call-by-value mechanism into a *call-by-reference* mechanism, and the argument is said to be *passed by reference*. FORTRAN always passes arguments by reference; C always passes by value with pointers used to simulate pass by reference.

Call by reference allows functions with input-modification side-effects and avoids the expense of copying the actual argument. To obtain the safety of call by value and the efficiency of call by reference, use a modified form of call by reference, *call by constant reference*. With call by reference, the formal argument refers to the actual argument. We are spared the cost of copying the argument object at the risk of unintentionally altering the argument object as a side-effect. This risk can be eliminated by declaring the formal argument to be const, a constant reference to the actual argument.

Let's look at an example. Suppose we want to write a function to find the index of the smallest element in an array of float elements (using a slightly modified version of SimpleArray<T> from Section 4.3; see Exercise 6.2). We might write the function like this:

ch5/args.C

```
int index_of_min(const SimpleArray1d<float>& a) {
    float min_val = a[0];          // Minimum value found thus far
    int   min_idx = 0;             // Index of minimum value found thus far
    for (int i = 1; i < a.numElts(); i++) {
        if (a[i] < min_val) {
            min_idx = i;
            min_val = a[i];
        }
    }
    return min_idx;
}
```

By making the formal argument a const reference, we have obtained the efficiency of call by reference with the safety of call by value. Moreover, the function declaration

ch5/args.C

```
extern int index_of_min(const SimpleArray1d<float>& a);
```

that would appear in a header file signals our intention that the actual argument of index_of_min not be modified. This intention is enforced by the compiler.

We summarize our discussion of call by reference and call by const reference in two guidelines:

- Assume that an argument passed by (non-const) reference is modified.

In fact, C++ also makes this assumption, and it will flag an error if a const object is passed with call by reference:

ch5/con-callbr.C

```
extern int f(int&);     // Non-const call by reference
f(3);                   // WRONG: const actual argument for non-const int& formal argument
```

For this reason, the potential FORTRAN error of overwriting literal constants is avoided in C++. It is also incorrect to pass an actual argument that would require a type conversion:

ch5/con-callbr.C

```
double d;
f(d);.  // WRONG: temporary required for non-const int& formal argument
```

Permitting this code would be confusing: f() would appear to modify its argument but would really modify the temporary object created during the conversion. Conversely, for arguments not intended to be modified,

- Pass small objects by value and large objects by const reference.

The choice between call by value and call by const reference is an efficiency issue and hence machine dependent: Generally nonarray built-in objects should be passed by value and all other arguments should be passed by const reference.

5.3.3 Array Arguments

C++ built-in arrays are passed by reference, even when the formal argument indicates pass by value. Consider the code:

ch5/arrayArg.C

```
double runningSum(float y[]) {
    // Replace y[i] with the sum of y[j], 0 <= j <= i
    double sum = y[0];
    for (int i = 1; i < 5; i++) {
        sum += y[i];
        y[i] = sum;
    }
    return sum;
}
```

When runningSum() is called in the code

ch5/arrayArg.C

```
float x[5] = {10., 20., 30., 40., 50.};
// ...
runningSum(x);
```

each element of the argument array is set to the sum of it and the preceding elements. There is no special rule that dictates this behavior of C++ built-in arrays passed as arguments; rather it is a consequence of an array variable acting as a pointer to the first element of the array (cf. Section 2.11). The function runningSum()

has a formal argument y of type float* (the declaration float* y would be equivalent). When runningSum() is called, the formal argument y is initialized from x converted to a float*. The assignment statement changes items in the array pointed to by y, the same array pointed to by x.

Input-modification side-effects can be prevented for C++ array arguments using const. For example, a function declared like

ch5/arrayArg.C

```
extern double g(const float y[]);
```

cannot alter the elements of its argument because the formal argument is (effectively) a pointer to const elements. If you use C++ arrays instead of an array class, remember to

- Declare C++ built-in array arguments const, unless the function is intended to alter the array elements.

5.3.4 Default Arguments

It is sometimes convenient to omit some of a function's actual arguments. Consider a function log_of() that computes the logarithm of a number to a specified base. It might be declared thus:

ch5/logof.C

```
extern double log_of(double x, double base);
```

But in scientific computation the base is often $e \approx 2.71828$, and it would be convenient to omit the second argument in these cases. This can be done with a *default argument*:

ch5/logof.C

```
#include <math.h>
extern double log_of(double x, double base = M_E);    // M_E in <math.h>
```

The = M_E specifies a value that is to be supplied by the compiler when the second actual argument is omitted: The call log_of(2) returns the logarithm base *e* of 2, and the call log_of(2, 10) returns the logarithm base 10 of 2. Even in the first instance, log_of() has two formal arguments: The second actual argument is supplied automatically.

There are a few syntactic matters to keep in mind when using default arguments. First, all arguments to the right of a formal argument with a default argument must also have a default argument. Therefore the declaration

ch5/logof.C

```
extern double f(int x = 3, double y);    // WRONG: y must have a default argument
```

is wrong. Second, the default argument must be present in the function declaration seen by the calling code, for it is the calling code that supplies the actual argument:

- Supply default arguments with the function's declaration in the header file, not with the function's definition in the implementation file.

5.3.5 Functions without Arguments

A function without arguments is declared with an empty argument list, like this:

ch5/args.C

```
extern float f();
```

If you read code written by a C programmer, you might also see a function without arguments declared with an argument list of (void), like this:

ch5/args.C

```
extern float f(void);
```

This second form is compatible with C but is otherwise confusing: f() takes no arguments, not a single argument of the fictitious type void. We recommend against using it in new C++ code.

5.4 Function Return Types

Every function has a return type and must return a value unless the return type is void, as in the following declaration:

ch5/retdcl.C

```
extern void display_greeting();
```

The value to be returned is given by a return statement in the function, and that value is used to initialize a hidden variable of the specified return type. As with passing an argument by value, this can result in copying the return value and/or converting it to a different type, and the same considerations apply. For example, in the code

ch5/args.C

```
float f() {
    // ...
    return 1;
}
```

the int return value is used to initialize a float variable—the function's return value—to 1.0.

Likewise an object can be returned by reference or by constant reference in a manner analogous to call by reference and call by constant reference. Using the index_of_min() function on page 128, we could write a function that returns a reference to the minimum element of an array:

ch5/args.C

```
float& min_elt(SimpleArray1d<float>& a) {
    return a[ index_of_min(a) ];
}
```

This technique is powerful, for it provides a way to access an object via an arbitrary computation. For example, the call to min_elt() in the code

ch5/args.C

```
SimpleArray1d<float> z(4);
// ...
min_elt(z) = 0;
```

returns a reference to the smallest element in the array. Using this returned reference on the left side of an assignment sets the smallest element of z to 0.

- Use return by reference or return by constant reference to provide access to a selected portion of an object or of a function argument passed by reference.

With this power comes the danger of returning a reference to an object that will cease to exist when the function terminates. For example, the objects associated with (non-static) local variables (see Section 2.16.4) are discarded when the function returns. Thus the following function is incorrect:

ch5/args.C

```
double& g() {
    double x;
    // ...
    return x;    // WRONG: reference to local variable returned
}
```

- Do not return (non-static) local variables or constants by reference.

Fortunately, many compilers are able to detect such errors.

5.5 Overloaded Functions

We saw in Section 5.2 that a function identifier can be overloaded so that one identifier refers to a group of functions that (presumably) accomplish the same task with different types of objects. When an overloaded function identifier is called, C++ uses the types of the actual arguments to select from among the several functions having the specified name. The best match wins. The precise

§13 rules that define best match are complicated, and we shall only sketch them here.

The idea of best match is formulated in terms of a function's formal argument types. The return type has no role. Indeed functions in the same scope that differ only in their return type are not allowed.

5.5.1 Single Argument Functions

We first consider the meaning of best match for single argument functions; the extension to multiple argument functions is discussed in Section 5.5.2. An *argument matching* process determines which function to call from among all the func-

tions having the specified identifier and declared in the same scope. The matching process considers converting the actual argument to the formal argument type for each eligible function. The conversions considered are ordered in four levels, from no or trivial conversion up to user-defined conversion. The matching process chooses the function that matches the earliest conversion rule in the sequence. If more than one function matches at any level, the function call is ambiguous and is not allowed; if no match is found, the function call is also not allowed. If a single, unambiguous function call is found, the compiler arranges for the appropriate conversion and function call.

The four major conversion rules, in order of decreasing preference, are as follows:

Exact Match or Match with Trivial Conversions If the types of the actual and formal arguments are identical, the arguments match. Given

ch5/abs.C

```
extern int      abs(int);
extern double abs(double);
extern double abs(complex);
```

the following function calls match exactly the function declarations indicated:

ch5/abs.C

```
int i = -3;
int iabs      = abs(i);      // Calls abs(int)
double d = 3.2;
double dabs = abs(d);      // Calls abs(double)
complex c(-1, 2);
double cabs = abs(c);      // Calls abs(complex)
```

If there is an exact match, it is the best match.

Argument types are also considered to be an exact match if only *trivial conversions* are required. The trivial conversions include conversions from a type T to T& and vice versa, and from a (built-in) array of T to a T*. Thus the declarations

ch5/abs.C

```
extern int      abs(int&);
extern double abs(double&);
extern double abs(complex&);
```

cannot coexist with the preceding declarations. Intuitively, the fact that conversions from T to T& and vice versa are considered trivial captures the idea that there is no reasonable way to choose between call by value and call by reference simply by looking at the function call.

On the other hand, call by reference implies that the function modifies its argument, while call by constant reference implies that the argument is not modified. If both call-by-reference and call-by-constant-reference versions of a function

exist and the actual argument is a const object, then it is clear that the call-by-constant-reference version should be invoked. Conversely, if the actual argument is a non-const object, the call-by-reference version should be invoked. This latter idea is captured by the rule that conversions from T& to const T&, or from T* to const T*, are considered worse than other trivial conversions. From the perspective of a function writer, this distinction between a formal argument of type T& and a formal argument of type const T& lets us provide pairs of functions, one that modifies its argument and one that does not.

Consider the min_elt() function from page 131. It returns a non-const reference to an array element, allowing that element to be modified. This precludes passing a const array to min_elt(). We can remedy this by adding the const half of the pair:

ch5/args.C

```
const float& min_elt(const SimpleArray1d<float>& a) {
    return a[ index_of_min(a) ];
}
```

As we shall see, used with member functions, this ability to distinguish between const and non-const objects is important for providing low-cost, read-only access to large objects.

Match with Promotions A sequence that consists of conversions from an integral type (char, short int, or enumeration) to an int, from a float to a double, or a trivial conversion is called a *match with promotion*. A match with promotion is better than all others except an exact match.

Matches with promotion are often used to distinguish between an integral argument and a floating point argument. For example, with the declarations

ch5/args.C

```
extern double exp(int y);
extern double exp(double y);
```

calling exp with a char, short int, or int argument invokes exp(int) (converting the argument to int if necessary), and calling exp with a float or double invokes exp(double) (again with the necessary conversion).

Match with Standard Conversions A sequence that consists of *standard conversions* (i.e., conversions built into C++) or trivial conversions is better than all others except a match with promotions or an exact match.

Consider a function that will work with either an integral or floating type of arithmetic argument. All arithmetic types (except long double) can be converted to double via a standard conversion. Therefore if the function is written with a double formal argument, it can be called with any arithmetic argument via match with standard conversions. Sometimes such a general function is provided in the early stages of a program's development, and more efficient or more accurate functions

are provided for specific types later as needed. The foregoing exp function declarations illustrate this: The double version could have been provided first and the int version added later. With the int version present, the preference for match with promotion over match with standard conversions ensures that the int version is invoked for all integral arguments.

Another form of standard conversion will be important in Chapter 9, where we discuss interface categories.

Match with User-Defined Conversions Finally, sequences involving *user-defined conversions* are examined. A user-defined conversion is a conversion to or from a class object. For example, a ComplexFloat class would probably define a conversion from float to ComplexFloat. (User-defined conversions are discussed in Section 6.4.) A sequence of conversions that consists of at most one user-defined conversion and of trivial conversions is better than all others except a match with standard conversions, match with promotions, or an exact match. To avoid surprising results, no more than one user-defined conversion is considered in any match.

5.5.2 Multiple Argument Functions

The best match for a function with multiple arguments is determined by looking at each argument individually. If one function provides a better match for one argument, and at least as good a match for the other arguments, then that function is called. If there is no such function, the call is in error. For example, given these declarations

ch5/args.C

```
extern void f2(int, float);
extern void f2(int, double);
```

the function call

ch5/args.C

```
double d = 2.0;
f2(1, d);    // Calls f(int, double)
```

invokes f(int, double) because the first argument matches exactly for both functions, but the second argument matches with standard conversion for the first function and exactly for the second function. Since the second function provides a better match for the second argument, it is considered the best matching of the two functions. On the other hand, the call

ch5/args.C

```
f2(1, 2);    // WRONG: ambiguous
```

is ambiguous because both functions match exactly on the first argument and match with standard conversion on the second argument; neither function meets the requirement of providing a better match for at least one argument.

5.5.3 Summary

As we said, the precise definition of best match is complex. If you find yourself needing to delve into the details, consider simplifying your design instead.

- Used simply, function overloading avoids needless repetition of function definitions for similar types of arguments. But overused or used in a way that depends on subtleties of the C++ rules, overloading is confusing.

5.6 Function Templates

Overloaded functions have the same name but differ in their function arguments. They allow us to express the similarity of action on different object types. When the similarity is stronger, to the point of identical code applied to different object types, C++ provides function templates. The sqr() function template of Section 4.4 is a simple example of this. In this section, we discuss function templates in more detail using a more sophisticated example.

The widely use BLAS Level 1 FORTRAN subroutine library [69, 40] has the four functions SAXPY, DAXPY, CAXPY, and ZAXPY, all of which implement $y \leftarrow \alpha x + y$ for vectors x and y and scalar α. From the caller's perspective, they differ only in name and vector element type. The BLAS documentation [40, Appendix A] lists a subroutine named _AXPY, where the underscore is replaced by one of the letters S (REAL), D (DOUBLE PRECISION), C (COMPLEX), or Z (COMPLEX*16) to indicate the element type. This family of subroutines would be expressed in C++ using a *function template declaration*, like this:

ch5/axpy.h

```
template<class EltType>
extern void axpy(EltType alpha, const Vector<EltType>& x, Vector<EltType>& y);
```

where Vector<T> is just a synonym for SimpleArray1d<T> (a slightly modified version of SimpleArray<T> from Section 4.3; see Exercise 6.2).

A function template declaration consists of the keyword template, followed by a comma-separated list of template arguments enclosed within angle brackets (< . . . >), followed by a function declaration. In this case, there is one template argument, class EltType, and axpy is said to be parameterized by EltType. The keyword class in the template argument indicates that EltType is a type, either a class type or a built-in type. The entire function template declaration specifies that there is a family of functions, each with the overloaded identifier axpy. Any function whose declaration can be obtained by substituting a type for EltType is a member of the family. Thus the following analogs of the Level 1 BLAS _AXPY subroutines are all members of the family

ch5/blastemp.C

```
void axpy(float       alpha, const Vector<float>&       x, Vector<float>&       y);
void axpy(double      alpha, const Vector<double>&      x, Vector<double>&      y);
void axpy(ComplexFloat alpha, const Vector<ComplexFloat>& x, Vector<ComplexFloat>& y);
void axpy(complex     alpha, const Vector<complex>&     x, Vector<complex>&     y);
```

but the function

ch5/blastemp.C

```
void axpy(float       alpha, const Vector<double>&      x, Vector<double>&      y);
```

is not, because no substitution for EltType yields the declaration.

A *function template definition* is a function template declaration with a body:

ch5/axpy.c

```
template<class EltType>
void axpy(EltType alpha, const Vector<EltType>& x, Vector<EltType>& y) {
    if (alpha != 0.0) {
        int n = x.numElts();
        for (int i = 0; i < n; i++) y(i) += alpha * x(i);
    }
}
```

One can think of a function template definition as a mold from which function definitions are stamped by the compiler as needed.

When function templates are declared, the C++ compiler takes three steps to resolve function calls:

1. If an exact match (without trivial conversions) of a nontemplate function is found, the matching function is called.
2. Otherwise, if a function template declaration that matches is found, it is used and the function definition is generated from the function template definition if necessary.
3. Otherwise, if a function is found using the ordinary argument matching process (Section 5.5), it is used.

If, at any step, more than one function is found, the call is ambiguous and considered an error. It is also an error if no function is found.

Function template argument matching (step 2 in the preceding list) is like function argument matching, except that only exact matching with trivial conversions is considered. Additional conversions having to do with pointers and references to class objects are also allowed; these are discussed in Section 12.3.

Here are some examples assuming that we have the aforementioned axpy function template definition (but not any other axpy declarations):

ch5/taxpy.C

```
Vector<double> a(4);
Vector<int> ai(4);
Vector<complex> ac(4);
Vector<double> b(4);
Vector<int> bi(4);
Vector<complex> bc(4);
// Initialize a, b, ai, and bi ...
axpy(10.0, a, b);       // Matches template with EltType = double
axpy(10, ai, bi);       // Matches template with EltType = int
axpy(10, a, b);         // Incorrect: no match
axpy(10.0, ac, bc);     // Incorrect: no match
```

Since there are no nontemplate function declarations, we move directly to the second step. The first two calls match the template by exact matches of the first and third arguments and trivial conversion (to const) of the second argument. The third call fails because 10 is an int, not a double, and standard conversions are not applied when matching function templates. Similarly, the fourth call fails because user-defined conversions (e.g., from double to complex) are not applied.

Now let's add in declarations for the aforementioned BLAS-like functions:

ch5/taxpy1.C

```
extern void axpy(float          alpha, const Vector<float> &       x, Vector<float> &       y);
extern void axpy(double         alpha, const Vector<double> &      x, Vector<double> &      y);
extern void axpy(ComplexFloat alpha,
                 const Vector<ComplexFloat> & x, Vector<ComplexFloat> & y);
extern void axpy(complex        alpha, const Vector<complex> & x, Vector<complex> & y);
```

Now matches are found for all four calls:

ch5/taxpy1.C

```
axpy(10.0, a, b);    // Matches template with EltType = double
axpy(10, ai, bi);    // Matches template with EltType = int
axpy(10, a, b);      // Matches axpy(double, const Vector<double> &, Vector<double> &)
axpy(10.0, ac, bc); // Matches axpy(complex, const Vector<complex> &, Vector<complex> &)
```

The first call fails to match exactly (without trivial conversions) any of the explicit function declarations, but it does match the function template. The second call matches the template, as before. The third and fourth calls each fail to match exactly any of the explicit function declarations or to match the template, but they each match an explicit function declaration using the ordinary argument matching process.

After a function declaration (template or nontemplate) is selected by the matching process, the compiler must find a corresponding definition. The rule is simple: If an explicit definition exists, it is used, regardless of whether or not

the matched declaration is a template; otherwise the function template definition is expanded with the appropriate parameters. For example, the aforementioned four calls result in three expansions of the axpy() function template, with parameter values double, int, and complex. If we were to supply an explicit definition for one of the functions, the compiler would use it instead of expanding the template definition for that case. This is called *template specialization*; see Exercise 5.5.

C++ uses the types of the actual arguments to select the types for expanding a function template declaration. Consequently, every template argument must be used in the function's formal argument types, and some kinds of commonality cannot be captured with function templates. For example, we cannot write

ch5/dot.C

```
template<class AccumType>   // WRONG: template argument not used in formal arguments
extern AccumType dot(const Vector<float>&, const Vector<float>&);
```

Since the template argument is not used in any of the formal arguments, and since the return type is not considered in the matching process, a call to dot does not provide the information needed for the compiler to choose a type for AccumType. Instead we must write separate functions:

ch5/dot.C

```
extern float    sdot(const Vector<float>&, const Vector<float>&); // Float accumulation
extern double ddot(const Vector<float>&, const Vector<float>&); // Double accumulation
```

5.7 Notes and Comments

5.1 Side-effects are often blamed for making programs obscure and prone to error [48, Section 6.2.1.1]. The presence of side-effects is the major difference between mathematical functions and programming language functions. Indeed, John Backus, the originator of FORTRAN, has proposed [9] that functional programming, a style of programming in which there are no variables and no side-effects, would turn our programs into mathematical objects we could reason about. (See [48, Chapter 8] for an informal introduction to functional programming.)

5.2 The rules for function templates are under debate in the ANSI C++ standardization committee. As specified in [44, Section 14.4c], a function template matches only with exact match, without trivial conversions. In practice, many have found these rules too restrictive. The rules we have described, which are from [74], are implemented in several commercially available compilers and are likely to be supported by the standard; further extensions are possible. The issues and various proposals are discussed in [47].

5.3 "All definitions are also declarations" is an oft-quoted but confusing bit of C++ syntactic wisdom. By the formal rules of the language, a program consists solely of declarations some of which are also definitions. Thus while virtually all nontrivial programs place declarations that are not definitions in header files and declarations that are definitions in implementation files, it is permissible and occasionally convenient

(especially for small test programs) to define a function that will be used locally without first declaring it separately. Such a definition must precede any call to that function in the file.

5.4 A function declaration specifies how the declared function interacts with code that calls it. Although we usually think of functions interacting with other code via arguments and returned values, throwing an exception is also an important form of interaction: A function that throws an exception does not return to the caller in the usual sense and will terminate the execution of a calling function that does not supply a catch for the thrown object.

The exceptions thrown by a function, either directly, by the function itself, or indirectly, by another function it calls, can be listed in a *throw specification* given as an optional part of the function declaration. The throw specification is not part of a function's type, and the compiler does not check that a function obeys its throw specification. Instead the check is done at runtime: If a function throws an exception not listed in its throw specification, a global, system-provided function is called. Its default action is to terminate the program. A function declared without a throw specification may throw any exception.

Exceptions are new to C++, and they were first supported in commercially available compilers in 1992. Therefore there is little experience to guide their use. In general, we have found exceptions to be useful and convenient. However, it is not clear to us that throw specifications, as presently defined, are useful. Initial versions of the code in this book had throw specifications; since we didn't perceive much benefit, we have removed them pending further experience.

5.8 Exercises

5.1 Given the order function from page 127, what would you expect that the code

ch5/swap.C

```
int x = 3;
int z = 4;
order(x, 2, z);
float y = 5;
order(x, y, z);
```

does? Try it.

5.2 What is wrong with the following code?

ch5/hasVal.C

```
int hasVal(int val, int n, int values[ ]) {
    // Return true if the n-element array "values" contains an element
    // with value "val"; otherwise, return false.
    for (int i = 0; i < n; i++) {
        if (values[i] = val) return 1;
    }
    return 0;
}
```

Change the formal argument declarations so that the compiler would detect this error.

5.3 The argument of index_of_min (page 128) is passed by constant reference, while the argument of min_elt (page 131) is passed by reference. Explain why.

5.4 Write function template declarations for some of the subroutines in a subroutine library that you use.

5.5 Write a function template definition that computes the dot product of two arrays. Then write a template specialization for arrays with float elements that accumulates in a double but returns a float result.

5.6 Given these function declarations

ch5/resolve

```
extern void f(float);
extern void f(double);
template<class T> extern void f(T);
```

explain how the following function calls are resolved:

ch5/resolve

```
f(1);
float x = 1.0;
f(x);
f('a');
f(1.0);
```

5.7 Define a family of classes with varying object sizes. Pass these varying-sized objects to two functions to investigate the relative performance of call by reference and call by value. For what size objects does call by reference become more efficient? Try your experiments with several compilers or on several machines. Does the crossover point vary?

5.8 Why doesn't the following code compile?

ch5/wrong1

```
extern double f(float);
extern double f(double);

void g() {
   f(1);
}
```

5.9 What is wrong with the following code:

ch5/wrong2

```
void f(int i) {
   cout << i << endl;
}

int main() {
   f(1.0e15);
   return 0;
}
```

CHAPTER 6

Functions and Classes

OBJECTS combine data and functions, with the data private to the objects and manipulated only by member functions. From the perspective of an object's *clients*, the other objects and functions that use the object, the member function declarations completely describe the object's behavior. The list of possible actions for each object is exactly the list of its members. Creatively and consistently selecting member function declarations increases the applicability of our classes, simplifies coding and altering the implementation of the functions themselves, and thus leads to better programs.

In this chapter, we focus on how member function declarations affect an object's behavior as viewed from the client's perspective. We begin the chapter by revisiting function overloading (cf. Section 5.5) in the context of member functions. Overloading is important in defining the behavior of an object from the client's perspective. Some member functions relate directly to an object's purpose: set a voltage, compute a value, etc. Others control the interaction of the objects with the larger C++ environment: how instances are created, copied, and destroyed; conversion to other kinds of objects; and assignment. These interactions are controlled by special member functions also discussed in this chapter.

6.1 Member Functions and Overloading

We saw in Section 5.2 that a function identifier can be overloaded so that it refers to a group of functions. When such an overloaded function identifier is used, C++ uses the types of the actual arguments to select one function from among the group, according to the argument matching process described in Section 5.5.

Member functions have similar behavior. If a class has more than one member function with the same identifier, but different argument types, the same argument matching process is used to choose among them. There is, however, one important difference: C++ distinguishes between const and non-const member functions, as illustrated by the following modified version of the Point class from page 87:

ch6/Point.h

```
class Point {
public:
    Point();                                // Create uninitialized
    Point(Number x, Number y);              // Create from (x,y)
    Number distance(Point point) const;     // Distance to another point
    Number distance(Line line) const;       // Distance to a line
    Number& x();                            // Reference to x-coordinate
    Number x() const;                       // Get x-coordinate
    Number& y();                            // Reference to y-coordinate
    Number y() const;                       // Get y-coordinate
private:
    Number the_x;                           // x-coordinate
    Number the_y;                           // y-coordinate
};
```

A member function is declared const by putting the keyword const immediately following the argument list. These member functions do not alter the value of any member data (regardless of whether the member data is public or private). This makes it clear to a human reader that these member functions are intended to access but not alter an object; it also instructs the compiler to enforce the intention, preventing future code changes from mistakenly altering objects. In this example, the const version of x() returns a copy of the node's x-coordinate, while the non-const version returns a non-const reference to it. The reference allows us to alter the value in the object; it would not be a legal return type for the const version of x().

When x() is called for a const instance, the const version is used; otherwise the non-const version is used:

ch6/UsePoint.C

```
Point p1(2, 3);
cout << p1.x() << endl;     // Uses non-const x()
p1.x() = 1;                 // Uses non-const x()
p1.y() = 2;                 // Uses non-const y()

const Point p2 = p1;
cout << p2.x() << endl;     // Uses const x()
p2.x() = 3;                 // WRONG: Operand of = must be an lvalue (Uses const x())
p2.y() = 4;                 // WRONG: Operand of = must be an lvalue (Uses const x())
```

Notice that C++ chooses between the const and non-const versions of member functions depending on whether or not the object for which they are called is const or non-const. The appearance of the function call on the left-hand side of an assignment operation is irrelevant.

Programs using large systems of classes and functions rely on const declarations to improve program correctness and efficiency. As described in Section 5.3.2, a function that takes a const reference argument can use large objects without copying them; equally important, a programmer observing the const declaration, can conclude that the function does not alter the values in the actual argument. This frees the programmer from creating a temporary copy to pass to a function "just in case" or from reading the function in detail, including calls to other functions to determine what data are altered.

To be able to use const objects,

- Declare member functions that are not meant to alter member data const.

In particular, using a const reference argument to signal that a function will not alter that argument object will be foiled unless the object's class supplies const member functions. For example, without the const version of `x()` it would not have been possible to print the x-coordinate of `p2`.

6.2 Initialization

Objects contain data and the data need initial values: Objects need initializers. These initializers are called *constructors* in C++; they consist of member functions specific to each class. Constructors are called, explicitly by the programmer's code or implicitly by the compiler, when an object is created.

Constructors are declared as member functions having the class name as function name. Constructors may not have a return type. As with other member functions, constructors can be overloaded. For example, given Point and LineSegment classes, we might write

ch6/Circle.h

```
class Circle {
public:
    Circle(double radius);                      // Of specified radius centered at origin
    Circle(Point center, double radius);        // From center and radius
    Circle(Point p1, Point p2, Point p3);       // From three points on circle
    Circle(LineSegment chord, Point p);         // From chord and point on circle
    Circle(LineSegment diameter);               // From diameter
    // ...
private:
    Point  the_center;
    double the_radius;
};
```

The overloaded constructors distinguish among different ways of initializing the object being created. Each of the constructors creates a circle from a different kind of geometric specification.

Constructors can be called explicitly, as in the expression `Circle(Point(1, 2), 0)`, or implicitly in variable declarations:

ch6/Circle.C

```
Point       p1(3, 4);        // Create Point p1 initialized to (3, 4)
Point       p2(5, 6);        // Create Point p2 initialized to (5, 6)
Point       p3(7, 8);        // Create Point p3 initialized to (7, 8)
LineSegment l1(p1, p2);      // Create LineSegment l1 with endpoints p1 & p2
Circle      ca(l1, p3);      // Create Circle with chord l1 through point p3
Circle      cb(l1);          // Create Circle with diameter l1
Circle      cc(p1, p2, p3);  // Create Circle through p1, p2, & p3
```

In each of these cases, a variable of the specified type is being declared; a constructor is called to initialize the object associated with the variable, with the constructor arguments enclosed in parentheses given after the variable name.

For consistency, this syntax can also be used to initialize variables of built-in type:

ch6/dcl-syntax.C

```
int i = 3;    // i is an int initialized to 3
int j(3);     // j is also an int initialized to 3
```

In fact, it is convenient to think of there being a single-argument constructor for each of the built-in types.

The parentheses must be omitted if there are no constructor arguments (or, for built-in types, initializer); otherwise you will be writing a function prototype:

ch6/dcl-syntax.C

```
int f1();        // f1 is a function returning int, taking no arguments
int f2;          // f2 is an int
Point p4();      // p4 is a function returning Point, taking no arguments
Point p5;        // p5 is a Point
```

Constructors initialize their object's data members. A constructor definition can supply a comma-separated list of *member initializers*, each a data member name and appropriate constructor arguments. Here, for example, are some of the `Circle` constructor definitions:

ch6/Circle.C

```
Circle::Circle(double radius) :
    the_center(0, 0),
    the_radius(radius) {
}
```

```
Circle::Circle(Point center, double radius) :
    the_center(center),
    the_radius(radius) {
}
```

The list of member initializers follows a colon (:) and precedes the opening brace of the function body. The function body is required even if it is empty. The specified arguments are passed to the named data member's constructor, as is done when initializing an ordinary variable. Notice that this syntax works for data members of both built-in and programmer-defined types.

The arguments passed to the member initializers can be arbitrary expressions:

ch6/Circle.C

```
Circle::Circle(LineSegment diameter) :
    the_center(diameter.midPoint()),
    the_radius(.5 * diameter.length()) {
}
```

The expressions used to initialize a data member can involve the values of other data members:

ch6/Circle.C

```
Circle::Circle(Point p1, Point p2, Point p3) :
    the_center(centerOfCircleThrough(p1, p2, p3)),
    the_radius(the_center.distance(p1)) {
}
```

(We assume that the function centerOfCircleThrough() computes the center of the circle through three Point objects.) The initialization expression for the_radius involves the value of the_center: the_center must be initialized before the_radius. Data members are initialized in the order the members are listed in the class declaration; the order of the member initializers in constructor definitions is irrelevant. We recommend the following:

- Avoid using the value of one member in the initialization for another member.

When efficiency or convenience dictate using the value of one member in the initialization of another member,

- Member initializers should appear in the order in which the members are declared.

Ignoring this recommendation yields code that looks correct, but isn't.

A member initializer does not need to be provided for a data member in three cases:

1. The data member is of a built-in type. The member is not initialized; its value is undefined.

2. The data member is of a programmer-defined type and the class provides a *default constructor*, a constructor that can be called without an argument. The member is initialized by calling the default constructor. For consistency, it is convenient to think of there being a default constructor for each built-in type that does nothing.

3. The data member is of a programmer-defined type and the class does not have any constructors. In this case, C++ initializes the data with a C++-generated default constructor. This generated function calls default constructors for the data members.

A member initializer must be provided for a data member in three cases:

1. There is no default constructor (either programmer provided or C++ generated) for the member's type.

2. The data member is a reference. A reference is a name for an existing object (cf. Section 2.15 or Section 3.9) and therefore must be initialized.

3. The data member is declared const. A const variable is a variable that, once initialized, can't be changed (cf. Section 2.8.1). Likewise, a const member can't be changed once initialized; if no initializer is provided, the default constructor is called. For built-in types, this leaves the member uninitialized.

6.3 Copying

Class objects can be copied by assignment or by initialization. Copying by assignment is copying the value of an object into an existing class object; copying by initialization is copying the value of an existing class object into an object as it is created. In C++ jargon, the word copy refers to copying on initialization. We discuss copy on initialization here and copy on assignment in Section 6.6.

When an object is passed by value to a function, it is copied; when an object is returned from a function, it is copied. For built-in objects, copying an object means copying its value. For classes that do not have a copy constructor declared, C++ generates a copy constructor that recursively copies member data and base objects. For class objects, C++ allows the programmer to define what copying means for instances of each class, by defining a *copy constructor*.

A copy constructor for a class `C` is a constructor that can be called with a single argument of type `C`. These include `C::C(const C&)` and `C::C(C&)`, as well as any multiple argument constructor beginning with a `const C&` or `C&` argument and having defaults for all remaining arguments (cf. Section 5.3.4).

In most ways, copy constructors are like other constructors: They are called to initialize a class object as it is created. They are different in that C++ calls the copy constructor when a class object is passed by value, returned by value, or thrown as an exception.

A copy constructor must be defined whenever the C++-generated one is inadequate, most commonly in a class that has built-in pointers as data members. Automatic copies of built-in pointers duplicate the pointer values, not the objects pointed to. This called a *shallow copy*. Programmer-defined copy constructors can copy the object pointed to if this is desired. This behavior is called *deep copy*. Chapter 14 introduces pointer classes that automatically supply deep-copy behavior.

For an example of a class without an explicit copy constructor, look at the Circle class defined on page 145. The C++-generated copy constructor copies the_radius and the_center. Since the_center is itself a class object, it is copied by calling the copy constructor for Point, be it C++generated or programmer supplied. The copy constructor can be called explicitly, as in the second line of this example:

ch6/Circle.C

```
Circle c1(Point(1, 2), 10);      // No copy
Circle c2(c1);                   // c2 initialized by copying c1
Circle c3 = 10.0;                // Create Circle(10.0) then initialize c3 from it
```

or it can be called implicitly, as on the third line (where a temporary Circle object is created and copied). Note that the = operator in this statement is not an assignment but rather is part of the initializer for c3. It has the following meaning: A constructor for the type of object being declared (Circle here) is called, with the object on the right-hand side of the = as its argument (10.0 here). The resulting object is then copied (not assigned) into the object being declared (c3 here) via its copy constructor.

The C++-generated copy constructor has the desired behavior for many, but not all, classes. Consider the SimpleFloatArray class from Section 4.2, repeated here for convenience:

ch6/SimpleFloatArray.h

```
class SimpleFloatArray {
public:
    SimpleFloatArray(int n);                                  // Create array of n elements
    SimpleFloatArray();                                       // Create array of 0 elements
    SimpleFloatArray(const SimpleFloatArray&);                // Copy array
    ~SimpleFloatArray();                                      // Destroy array
    float& operator[ ](int i);                                // Subscripting
    int numElts();                                            // Number of elements
    SimpleFloatArray& operator=(const SimpleFloatArray&);     // Array assignment
    SimpleFloatArray& operator=(float);                       // Scalar assignment
    void setSize(int n);                                      // Change size
```

```
private:
    int num_elts;                                   // Number of elements
    float* ptr_to_data;                             // Pointer to built-in array
                                                    // of elements
    void copy(const SimpleFloatArray& a);           // Copy in elements of a
};
```

The C++-generated copy constructor would copy the pointer ptr_to_data and the integer the_num_elts; none of the elements would be copied. Copying a SimpleFloatArray object would leave us with two array objects but only one set of elements. Instead the copy constructor we supplied copies the count, creates a new built-in array to hold the copied elements, and then copies the elements:

ch6/SimpleFloatArray.C

```
SimpleFloatArray::SimpleFloatArray(const SimpleFloatArray& a) {
    num_elts = a.num_elts;
    ptr_to_data = new float[num_elts];
    copy(a);    // Copy a's elements
}

void SimpleFloatArray::copy(const SimpleFloatArray& a) {
    // Copy a's elements into the elements of *this
    float* p = ptr_to_data + num_elts;
    float* q = a.ptr_to_data + num_elts;
    while (p > ptr_to_data) *--p = *--q;
}
```

Since shallow or pointer-value copying is the behavior of C++'s built-in pointers,

- A class that has built-in pointer member data not referring to data shared with other objects should have a copy constructor.

Alternatively, the built-in pointer can be replaced by a programmer-defined pointer that does deep copies; see Chapter 14.

With rare exception, copying an object doesn't change it. Therefore

- The argument to a copy constructor should be a const reference.

6.4 Conversion

We expect to be able to convert an int to a float, a float to a double, and so forth. The same capability is necessary for class objects; for example, given a Complex class, we expect to be able to convert any numeric type to Complex. Conversion of class objects can be specified by single argument constructors or by conversion operator member functions.

6.4.1 Constructors as Convertors

Initializing an object with a single argument constructor can be viewed as converting from an object of the argument type to an object of the class type. C++ supports this view: A constructor that can take a single argument defines a user-defined conversion. Such conversions are used in the argument matching process (Section 5.5) and in any other context in which C++ does automatic type conversion. Consider the following Boolean class that represents true and false values:

SciEng/Boolean.h

```
class Boolean {
public:
    // Constants
    enum constants { false = 0, true = 1 };

    // Construction.
    Boolean()                       {} // Construct uninitialized.
    Boolean(int i) :        v(i != 0)  {} // Construct and initialize to (i != 0).
    Boolean(float f) :      v(f != 0)  {} // Construct and initialize to (f != 0).
    Boolean(double d) : v(d != 0) {} // Construct and initialize to (d != 0).
    Boolean(void* p) :   v(p != 0) {} // Construct and initialize to (p != 0).

    // Conversion.
    operator int() const { return v; } // To allow "if (boolean-value)..."

    // Negation.
    Boolean operator!() const { return !v; }
private:
    char v;
};
```

Boolean objects can be initialized by a variety of built-in objects. A void* is a pointer to an object of unknown type; any pointer can be converted automatically to a void* in a manner that converts a null pointer to a null void* and a non-null pointer to a non-null void*. The conversion operator, operator int(), is described in Section 6.4.2 and the negation operator is discussed in Section 6.5.

Function argument matching, as described in Section 5.5, selects the constructor called when a Boolean is created. Here are some examples:

ch6/Bool-examples.C

```
Boolean b1(Boolean::true);      // Calls Boolean(int)
Boolean b2(3);                  // Calls Boolean(int)
int* p_i = new int(3);
Boolean b3(p_i);                // Calls Boolean(void*)
Boolean b4(3.0);                // Calls Boolean(float)
```

The Boolean object created is initialized to true if the constructor argument is nonzero and false otherwise. Each of the single argument constructors defines a conversion from a built-in type to Boolean. In the function

ch6/Bool-examples.C

```
Boolean has_real_solution(double a, double b, double c) {
    // Does ax**2 + bx + c = 0 have a real solution?
    return sqr(b) >= 4 * a * c;
}
```

the >= operator yields an int result, either 0 or 1. To return a Boolean, C++ uses the Boolean(int) constructor to convert the int to a Boolean object to be returned. More precisely, C++ creates a temporary Boolean object, which it initializes using the constructor.

6.4.2 Conversion Operators

Constructors provide conversion from an external type to the class being constructed. It is also necessary to be able to do conversions in the opposite direction, from a class to another type. For example, we want to be able to use Boolean objects as logical values in if statements, while loops, etc., which interpret a zero value as false and a nonzero value as true. Doing this requires that we be able to specify how Boolean objects should be converted for logic testing. A *conversion function* is a member function with the target type name following the keyword operator. The Boolean class (page 151) has a conversion function to int. Neither argument nor return type can be specified for a conversion function: The int conversion function must return an int object; as a member function, it operates on the object for which it is called.

Like a single argument constructor, a conversion function specifies a user-defined conversion, considered in argument matching and in other contexts in which C++ converts from one type to another. In the code

ch6/Bool-examples.C

```
if ( has_real_solution(a, b, c) ) {
    // ...
    return 1;
}
```

the function has_real_solution() returns a Boolean object; C++ calls the user-defined conversion from Boolean to int to yield an int object; that object is then tested for a zero (null) or nonzero value.

6.4.3 Implicit Conversions and Ambiguity

As the Boolean class illustrates, user-defined conversions are a means of integrating a new type into the built-in type structure of C++. The objects in the

new Boolean type can be constructed from int, void*, and double objects; they can be converted to int objects. Conversions are also needed among programmer-defined types. We need to be able to convert a ComplexFloat to a ComplexDouble. Either ComplexDouble should have a constructor that takes a ComplexFloat, or ComplexFloat should have a conversion function to ComplexDouble. Sometimes the choice is determined because we can't modify one of the classes. For example, many compilers come with a double precision complex class called complex: We might equate ComplexDouble with complex using a typedef. Unable to change the complex class to add a constructor for ComplexFloat, we would then choose to provide a conversion function in ComplexFloat.

In most cases, there is no reason to prefer a constructor over a conversion function or vice versa.

- Provide conversion from one class to another with either a constructor or a conversion function, but not both.

Violating this guideline will lead to ambiguity:

ch6/ambig-convert.C

```
class ComplexDouble;

class ComplexFloat {
public:
    // ...
    operator ComplexDouble() const;
    // ...
};

class ComplexDouble {
public:
    // ...
    ComplexDouble(const ComplexFloat&);
    // ...
};

ComplexFloat cf;
// ...
ComplexDouble cd = cf;    // WRONG: ambiguous conversion
```

Here the compiler cannot choose between ComplexFloat conversion to ComplexDouble and construction of a ComplexDouble from a ComplexFloat object.

Conversions can also introduce ambiguities in more subtle ways. Let's fix the preceding ambiguity and add functions for computing arg z and a ComplexInt class with conversions to both ComplexFloat and ComplexDouble:

ch6/ambig-convert2.C

```
class ComplexFloat;
class ComplexDouble;

class ComplexInt {
public:
    // ...
};

class ComplexFloat {
public:
    // ...
    ComplexFloat(ComplexInt);
    // ...
};

class ComplexDouble {
public:
    ComplexDouble(ComplexInt);
    ComplexDouble(ComplexFloat);
    // ...
};

extern float    arg(ComplexFloat z);      // Principal value of the argument of z
extern double arg(ComplexDouble z);    // Principal value of the argument of z
```

This all seems reasonable until we call arg on a ComplexInt:

ch6/ambig-convert2.¶

```
ComplexInt ci;
// ...
double a = arg(ci);    // WRONG: call matches arg(ComplexFloat) and arg(ComplexDouble)
```

The compiler wants to convert ci to either a ComplexFloat or a ComplexDouble so that it can call arg, but it has no way to choose between the two possible conversions. This kind of problem can arise whenever there are two or more conversions from a given type; the number of opportunities for ambiguities increases rapidly with the number of conversions.

- Minimize the number of user-defined conversions from any given type.

This kind of ambiguity is difficult to control because working code can be broken by introducing a new class. For example, there would be no problem if we only had the ComplexInt and ComplexFloat classes and the arg(ComplexFloat) function.

Previously working code could then be broken by adding the ComplexDouble class and arg(ComplexDouble) function. When this happens, there are three solutions:

- Remove one or more user-defined conversions to eliminate ambiguity.
- Explicitly disambiguate the offending call, like this:

  ch6/ambig-convert2.C

  ```
  double b = arg( ComplexDouble(ci) );
  ```

- If there are many ambiguous calls, introduce a function that disambiguates the call:

  ch6/ambig-convert2.C

  ```
  inline float arg(ComplexInt z) { return arg( ComplexFloat(z) ); }
  ```

  Introducing this function eliminates the need for an implicit conversion and hence eliminates the ambiguity.

Were arg a member function instead of a global function, there would be no ambiguity. (Why?)

Ambiguities induced by conversions cause compile errors. Conversions can also cause unexpected runtime behavior. Consider the following class and code fragment:

ch6/arrayAssign.C

```
class Array {
public:
    Array(int n);                       // Create n element array of int's
    Array(const Array&);                // Copy array
    Array& operator=(const Array&);     // Assign array
    // ...
};
// ...
```

ch6/arrayAssign.C

```
Array a(5);
a = 5;
```

This code compiles and executes without error. At first glance, the assignment statement appears to assign 5 to each of the elements of a. Closer inspection, however, reveals that the int 5 is being converted to an Array via the Array constructor, and then the newly created Array is assigned to a. The result is that the elements of a are set to garbage values.

In this particular case, the problem is easy to solve by adding an assignment operator that takes an int argument. Then no user-defined conversion is required, which is always a better match than when a conversion is required. In general,

- Avoid single argument constructors when possible.

The upshot is that while user-defined conversions can be valuable for integrating new types into the type system, they can also be a source of compile-time ambiguities and even unexpected runtime behavior. We therefore recommend the following:

- Provide few user-defined conversions.

Member functions can be called explicitly to do conversions without introducing the ambiguities of user-defined conversions. For example, ComplexInt could be modified to provide a single user-defined conversion to ComplexFloat and an ordinary member function for conversion to ComplexDouble:

ch6/ambig-convert3.C

```
class ComplexFloat;
class ComplexDouble;

class ComplexInt {
public:
    // ...
    ComplexDouble toComplexDouble() const;
};

class ComplexFloat {
public:
    // ...
    ComplexFloat(ComplexInt);
    // ...
};

class ComplexDouble {
public:
    ComplexDouble(ComplexFloat);
    // ...
};

extern float    arg(ComplexFloat z);      // Principal value of the argument of z
extern double arg(ComplexDouble z);     // Principal value of the argument of z

ComplexInt ci;
// ...
float a = arg(ci);
double b = arg( ci.toComplexDouble() );
```

6.4.4 Example: Fallible<T> Objects

Conversions can be used to define objects that add state and checking to existing objects. As an example, we develop a class template, Fallible<T>, that adds a binary state—valid or invalid—and some associated checking to an object of type T. If a Fallible<T> is in the valid state, it can be used as a T; if it is in the invalid state, attempting to use it as a T throws an exception.

Consider the problem of searching a set of nonoverlapping intervals for an interval that contains a specified value. We might represent an interval by the class

ch6/demoFal.C

```
class Interval {
public:
    Interval();
    Interval(double lo, double hi);
    double lo() const;
    double hi() const;
private:
    double the_lo;
    double the_hi;
};
```

and a set of intervals by the (simplified) class

ch6/demoFal.C

```
class SetOfIntervals {
public:
    SetOfIntervals(int max_number_of_intervals);
    void add(Interval);                                  // Add interval to set
    Fallible<Interval> intervalContaining(double val);   // Find interval containing val
private:
    int num_intervals;
    SimpleArray<Interval> intervals;
};
```

The member function intervalContaining() returns a Fallible<Interval>. This object is in the valid state if the set has an interval that contains the specified value and is otherwise in the invalid state.

Interval searches can then be done like this:

ch6/demoFal.C

```
SetOfIntervals s(5);
s.add(Interval(-100, -50));
s.add(Interval(-1, 1));
s.add(Interval(2, 4));
// ...
```

```
Interval interval_about_0 = s.intervalContaining(0);
Interval interval_about_5 = s.intervalContaining(5);   // Exception thrown
```

A fallible object's state can also be tested (without causing an exception)

ch6/demoFal.C

```
if ( s.intervalContaining(11.2).valid() ) {
    // ...
}
```

or a default value that is to be used when the object is in the invalid state can be provided:

ch6/demoFal.C

```
Interval HugeInterval(-1.0e10, 1.0e10);
Interval interval_about_10 = s.intervalContaining(10).elseDefaultTo(HugeInterval);
```

Fallible<T> uses both implicit and explicit conversion:

ch6/demoFal.C

```
template<class T>
class Fallible {
public:
    Fallible(const T& t) : is_valid(Boolean::true), instance(t) {}          // Valid.
    Fallible()           : is_valid(Boolean::false)           {}          // Invalid.
    Boolean failed() const { return !is_valid;                }  // True if invalid.
    Boolean valid()  const { return is_valid;                 }  // True if valid.
    void invalidate()      { is_valid = Boolean::false;       }  // Make invalid.
    operator T()     const;
    T elseDefaultTo(const T& default_value) const;   // Value if valid, else default_value

    class UsedInInvalidStateErr {
        // ...
    };
private:
    Boolean is_valid;
    T instance;
};
```

A Fallible<T> is created in the valid state when initialized with a T and in the invalid state otherwise:

ch6/demoFal.C

```
Fallible<Interval> SetOfIntervals::intervalContaining(double val) {
    for (int i = 0; i < num_intervals; i++) {
        Interval& t = intervals[i];
        if (t.lo() <= val && val <= t.hi()) return t;     // Valid state
    }
    return Fallible<Interval>();     // Invalid state
}
```

We can use a Fallible<T> as a T (as shown earlier) because it provides conversion to T; the conversion member function checks the state and either throws an exception or returns (a copy of) its T value:

ch6/demoFal.C

```
template<class T>
inline
Fallible<T>::operator T() const {
    if (failed()) throw UsedInInvalidStateErr();
    return instance;
}
```

Fallible<T> is a useful return type for functions that conceptually must return either a T or an indication that there is no T to return. Searching is a common situation like this: The search function either returns what is being requested or it must indicate that no such item was found. Often, failing to find the requested item is not really an error, so throwing an exception is inappropriate. On the other hand, the function must return some object but an attempt to use that object should fail.

- Fallible <T> is a useful return type for search functions.

6.5 Operator Functions

Operators are the symbols used in programs to specify operations on operands. Most programming languages provide operators for their built-in types, as in the FORTRAN expression X+Y. We have already discussed the operators available for C++ built-in types in Chapter 2, and we introduced operator members for the SimpleArray example in Section 4.3; Chapter 10 explores operators in a systematic fashion. In this section, we discuss how to specify operators for user-defined types.

Operators can be viewed as nothing more than a shorthand for function calls. But we don't write X+Y as ADD(X, Y) because operator notation mimics mathematical notation, a notation that has evolved to be compact yet expressive. We understand X*Y+A*B more quickly than ADD(MULT(X,Y), MULT(A,B)). The importance of convenient notation should not be underestimated.

C++ extends the concept of user-definable operators beyond arithmetic and even beyond logical operators to include additional symbols. Almost all symbols usable in C++ statements can be defined for programmer-defined types. Some of these symbols have meaning only in the context of programming, and deciding whether and how to use them for new types requires judgement: Does using the operator symbol clarify or obscure the meaning of the code? Remembering all of the abbreviations—operator definitions—may be harder than reading the more lengthy function calls.

Both operators and functions perform some action: Operators apply operations to operands; functions apply operations to arguments. Both operators and functions can return results, typically a transformation or combination of their inputs. C++ exploits this close connection between operators and functions to specify the effect of operators applied to class objects in terms of equivalent functions. Thus operators in C++ are specified (declared) and implemented (defined) in terms of equivalent functions, but they are used as operators.

The function name for an operator is formed from the keyword operator and the symbol for the operator; the operator's operands become the function's arguments. For example, a class called Complex might define an operator *= like this:

ch6/cmplx2.C

```
Complex& Complex::operator*=(const Complex& rhs) {
    float original_real_part = real_part;
    real_part = real_part * rhs.real_part - imag_part * rhs.imag_part;
    imag_part = imag_part * rhs.real_part + original_real_part * rhs.imag_part;
    return *this;
}
```

For consistency with the behavior of built-in operators, a Complex& is returned. Compare this definition to the use of *=:

ch6/cmplx2.C

```
Complex a(1,1);
Complex b(2,3);
a *= b;
```

The keyword operator appears in the declaration and definition, but not in the use. An operator function can also be called explicitly:

ch6/cmplx2.C

```
a.operator*=(b);
```

This call is equivalent to the preceding use of *=.

All C++ operators shown in Table 2.7 (page 31) can be defined by the programmer for class objects, except for member selection (.), member function dereference (.*), scope resolution (::), sizeof, and conditional (?:) operators. No new operator symbols can be invented, even by concatenation of existing operator symbols. For example, one cannot define an operator**() to provide exponentiation.

§3.4 The number of operator operands or the number of function arguments is called the *arity* of an operator or function. For example, the / operator is a binary—bi-arity—operator; the logical negation operator (!) is unary. All C++ operators, except the conditional operator (?:), are either unary or binary, with the arity fixed by the language. A few operators, −, +, &, and *, may be either unary or binary. In addition to fixing arity, the precedence and associativity of programmer-defined operators follows that of the built-in operators (see Table 2.7).

These restrictions force a syntactic correspondence between our expectations for operator use and the possible user-provided definitions. Operators shorten expressions; short expressions are easier to read—if the operators are understood:

- Give operators conventional definitions.

A corollary: The relations between built-in operators should be maintained with programmer-defined operators. For example, `a *= b` should bear the same relationship to `a = a * b` for class objects as it does for built-in objects.

A unary operator can be defined by either a member function with no argument or a global function with one argument; a binary operator can be defined by either a member function with one argument or a global function with two arguments. For example, earlier we defined the binary operator *= for Complex with a member function taking one argument. The left operand becomes the object for which the member function is invoked, and the right operand becomes the function argument. Contrast this with the following global function definition for the binary * operator: **ARM §1**

ch6/cmplx2.C

```
Complex operator*(const Complex& lhs, const Complex& rhs) {
    Complex result(lhs);
    return result *= rhs;
}
```

The operands lhs and rhs, both const references to Complex, appear as two arguments in the function declaration, and the sum object corresponds to the returned object of the operator function. (The return statement in this function exploits the return type of the operator*=() defined earlier.)

A global operator function must have at least one class object argument to avoid changing the meaning of C++ for built-in objects. There is no such requirement for the argument of a member operator function because, as a member function, the operator function will necessarily be invoked for a class object.

There is an important distinction between operators defined as member functions and operators defined as global functions: Since a member function is invoked on a class object, no conversion is done for the first operand of an operator defined as a member. For example, given the class definition

ch6/cmplx.C

```
class Complex {
public:
    Complex(float = 0.0);
    Complex(float, float);
    Complex operator+(const Complex&) const;
    // ...
};
```

the code

ch6/cmplx.C

```
Complex a;
Complex b;
b = a + 3.0;    // 3.0 is converted to Complex(3.0)
b = 3.0 + b;    // WRONG: + not allowed between double and Complex
```

will not compile. The expression 3.0 + b is effectively equivalent to the nonsensical expression (3.0).operator+(b). On the other hand, if we use a global function to define addition, the expected conversion is done:

ch6/cmplx1.C

```
class Complex {
public:
    Complex(float = 0.0);
    Complex(float, float);
    // ...
};

Complex operator+(const Complex&, const Complex&);

Complex a;
Complex b;
b = a + 3.0;    // 3.0 is converted to Complex(3.0)
b = 3.0 + b;    // 3.0 is converted to Complex(3.0)
```

The expression 3.0 + b is effectively equivalent to operator+(3.0, b); the usual function argument matching rules cause 3.0 to be converted to Complex(3.0), and operator+() is called. The desire for conversion of the left operand of binary operators generally dictates the following:

- Declare symmetric binary operators as global functions.

On the other hand, some binary operators, including all of the assignment operators (=, +=, etc.), are inherently asymmetric (if used in a manner analogous to their built-in definitions) in that they modify their left operand. Conversions are undesirable in such cases:

- Define asymmetric binary operators and unary operators that modify their operand as member functions.

The built-in binary operators come in pairs: a symmetric version and an asymmetric assignment version. Consistency with the language dictates that programmer-defined versions of these operators come in similar pairs.

- Define symmetric binary operators to call their corresponding asymmetric assignment operator.

For example, define operator*() to create a temporary variable initialized by the left operand, and then use the corresponding asymmetric operator to update the temporary variable, which is then returned. This is how we implemented Complex::operator*() on page 161. Chapter 10 explores a technique for applying these relationships systematically.

6.6 Assignment

When an object is assigned to a class object of type C, a function of the form C::operator=() is called. If no operator=() is declared for class C, then C++ generates an assignment operator that takes a const C& argument and assigns members recursively. The recursion terminates with the assignment of built-in objects, which are assigned as usual. C++ will not generate an assignment operator if a member cannot be assigned. For example, if the assignment operator for a member is private, the member cannot be assigned and consequently no assignment operator will be generated. **ARM §12**

A C++-generated assignment operator for a class C always expects a right operand of type C (including possible conversions). If you want to assign objects of other types, you must supply an operator=() member function that takes an argument of the type you want to assign, as we did in the SimpleFloatArray class on page 149.

Although copying and assignment are different, similar considerations apply; consider again the SimpleFloatArray class. If we had omitted the assignment operators and allowed C++ to generate one, assigning one SimpleFloatArray object to another would amount to assigning the pointer ptr_to_data and the integer the_num_elts; none of the elements would be assigned, and the assignment would leave us with two array objects but only one set of elements. Instead the assignment operator we supplied copies the count and the elements into the left-hand side:

ch6/SimpleFloatArray.C

```
SimpleFloatArray&
SimpleFloatArray::operator=(const SimpleFloatArray& rhs) {
    if ( ptr_to_data != rhs.ptr_to_data ) {
        setSize( rhs.num_elts );
        copy(rhs);
    }
    return *this;
}
```

The test handles the case in which an array is assigned to itself. Without the test, setSize() might delete the elements of the left operand, which would also be the elements of the right operand, before the values are copied.

- Assignment member functions should work correctly when the left and right operands are the same object.

Assignment operators almost always accompany explicit copy constructors:

- A class that has built-in pointer member data that are not meant to point to shared data should have an assignment operator.

Again, the pointer classes in Chapter 14 perform this automatically.

With rare exception, copying an object doesn't change it. Therefore

- The argument to an assignment operator should be a const reference.

We generally define assignment for basic components of programming like array and pointer classes (see Chapter 14), and then use C++-generated assignments elsewhere.

The built-in assignment operator yields a reference to its left-hand operand as its value. Thus for built-in objects one can write

ch6/mult-assign.C

```
a = (b = c);    // Assign c to b, then b to a
a = b = c;      // Assign c to b, then b to a
```

Consistency with C++ suggests the following:

- Assignment operator functions should return a reference to their left operand.

6.7 Special Operators

Some C++ operators have special constraints or meaning. Assignment (=), function call (()), subscripting ([ ]), and class member access (->) must all be nonstatic member functions since they all must operate on a class object. We discussed assignment in the preceding section; we survey the other three operators in this section.

■.4 The function call operator must also be a member. The name of this operator is operator()(); that is, the empty parentheses are in the position of the * for operator*(). For an object x, the expression x(5, 6) matches operator()(int,int): The number and types of the actual arguments in the function call select the possibly overloaded function call operator. The resulting expression looks like a function call:

- Use the function call operator for functionlike objects.

Chapter 13 uses function call operators for generalized array subscripting, treating arrays as functions of integer arguments. Chapter 17 describes general functionlike objects.

The square-bracket or subscripting operator, operator[ ]() is called for expressions like a[i] when a is a member of a class with an operator[ ]() defined. Only one argument can be specified for this operator.

- *Warning*: a[i,j] is a valid expression, but it calls operator[ ]() with the result of evaluating the expression i, j.

For this reason, we prefer to call this operator a bracket or projection operator; we use it for array projections in Chapter 13.

Class member access, operator->(), must also be a member, and it must return a pointer to an object, or something that acts like a pointer to an object.

- Use operator->() to create pointerlike classes.

The expression p_x->m will follow a pointer to get to m. If p_x is an object with an operator->(), that function is called and the pointer it returns is followed. This procedure continues until a built-in pointer is followed to the member m. Chapter 7 illustrates the use of operator->() in several pointer class examples.

The increment and decrement operators ++ and -- are unique in that they are available in both prefix and postfix forms. To distinguish between the two forms, the postfix form is treated as if it had a second operand, int(0). Thus the member operator++() is used for the prefix form (e.g., ++x) and the member operator++(int) is used for the postfix form (for example, x++); the analogous global functions are operator++(X&) and operator++(X&, int).

6.8 Destruction

A member function of a class C with the name ~C is a *destructor*. A destructor has neither arguments nor return type. Destruction is in some sense the opposite of initialization: If a class has a destructor, it is run whenever one of its instances is to be destroyed, either because the variable the instance is associated with goes out of scope, a compiler-generated temporary object is no longer needed, or delete is invoked on a pointer to the instance. Destructors are most often used to delete memory that was allocated during the object's construction. As such, the presence or absence of a destructor is an implementation decision that does not affect the object's interface.

ARM §1

6.9 Static Member Functions

Ordinary member functions manipulate instances of their class: They must be called with an instance, and they access and manipulate the instance's data members. Classes can also provide *static member functions*, member functions that are not called with an instance of the class. A static member function declaration is a member function declaration preceded by the keyword static; static member functions may not be declared const because they are not called with an object.

Static member functions are useful for constructing objects without introducing undesired conversions. Consider another version of ComplexFloat:

ch6/CmplxFlt.C

```
class ComplexFloat {
public:
    ComplexFloat(float r, float i);   // Construct from real & imaginary parts
    static ComplexFloat fromFile(istream& input = cin);
    // ...
};
```

The member fromFile() is a static member function that reads from an input stream, creates a ComplexFloat, and returns it:

ch6/CmplxFlt.C

```
ComplexFloat ComplexFloat::fromFile(istream& input) {
    float r;
    input >> r;
    float i;
    input >> i;
    return ComplexFloat(r, i);
}
```

Notice that the keyword static is not used in the function definition. From outside the class, a static member function must be called using the double-colon scope qualifier (::), like this:

ch6/CmplxFlt.C

```
ComplexFloat c = ComplexFloat::fromFile();
```

Static member functions are also useful as a naming mechanism. Consider the following class:

LapackWrap/BlasSubroutines.h

```
class Blas1Subroutines {
public:
    static double xdot(int n, const double* xp, int incx, const double* yp, int incy);
    static float  xdot(int n, const float* xp, int incx, const float* yp, int incy);
    static double xscal(int n, const double alpha, const double* xp, int incx);
    static float  xscal(int n, const float alpha, const float* xp, int incx);
```

```
    static double xnrm2(int n, const double* xp, int incx);
    static float  xnrm2(int n, const float* xp, int incx);
    // ...
};
```

It has no data members and only static members. The static member functions are like global functions grouped together and named with a prefix of Blas1Subroutines. Grouping these functions into the Blas1Subroutines class avoids possible name clash—these functions are only called if preceded with the scope qualifier, as in Blas1Subroutines::xdot.

6.10 Friend Functions

Ordinarily, nonpublic members of a class can only be accessed by that class's member functions. However, a class can grant such access to global functions and member functions of other classes. Friendship is granted by a friend declaration contained in the definition of the granting class. Here are some examples:

ch6/friend.C

```
class A;
class B {
    double g() const;
    double g();
    double g(int);
    void h();
};

class C {
public:
    friend int f(double);           // Global function: int f(double)
    friend double B::g() const;     // Member of B: double g() const
                                    // Note: only one of B's g members is a friend
    friend class D;                 // All members of D
    friend A;                       // All members of A; A already declared
private:
    friend void B::h();             // Member of B: void h()
};
```

There are several syntactic matters to keep in mind. Declaring a function to be a friend grants friendship to that one function, not to all functions of that name. In the preceding example, friendship has been granted to only one of B's three g() member functions. Friendship can be granted to all member functions of a class by granting friendship to the class. Indeed this is the only way to grant friendship

ARM §

to a member function of a class that has not been defined (in other words, a friend declaration cannot be the first mention of a member function). If the class has not been declared previously, add the class keyword to the declaration, as in the friend declaration for D. In this case, the class named is declared in the same scope as the class granting friendship:

ch6/friend.C

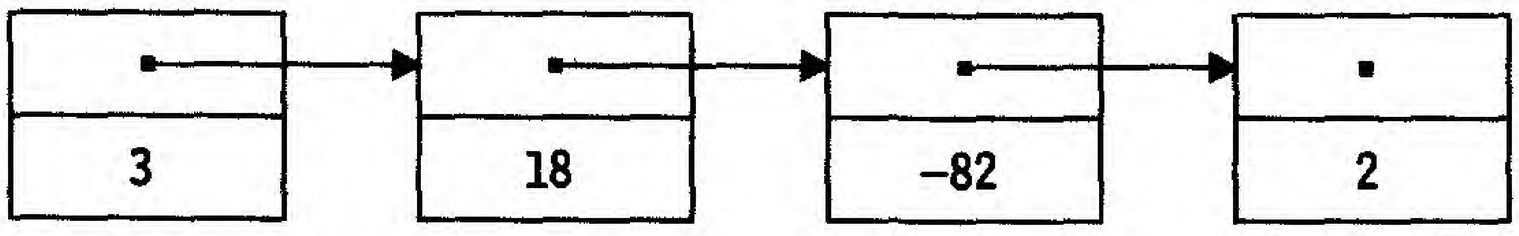
```
class X {
public:
    class Y {
    public:
        friend class Z;    // Declares X::Z
    };

    class Z {  };
};
```

Finally, the position of a friend declaration in a class definition is irrelevant: Access specifiers like private and public have no effect on friend declarations.

Let's consider a realistic example. Suppose we need to read from a file an unknown number of numeric values, normalize them by subtracting the minimum value, and write the normalized values. To compute the minimum we need to see all of the values; to normalize the values we need to know the minimum. Thus we need to read all of the values, computing the minimum as we go, and then normalize all of the values. We can't store the values in an array because we don't know how big to make the array; instead we use a *linked list*.

(If you know about linked lists, skip this paragraph.) Like an array, a linked list holds an ordered set of items. Unlike an array, inserting new items between items on a list or deleting items in the middle of a list does not need to move large numbers of items. Consequently, the number of items on a list need not be known at compile time, and the number of items can increase and decrease as the program runs. Each item is held in a *node*, which also has a pointer, called the *link*, to the next node. For example, a linked list containing the items 3, 18, −82, and 2 would be represented like this:



Each node contains a link; the last node's link is null to indicate the end of the list. By holding a pointer to the first node, all of the nodes, and hence all of the items, can be accessed; by keeping a pointer to the last node, new nodes can be added efficiently.

Two classes are necessary for a linked list holding items of type T: one, List<T>, to implement the linked list itself and the other, ListIterator<T>, to provide access to the items stored in the linked list. With these classes, our example program could be implemented like this:

ch6/demoList.C

```
int main() {
    // Read list of values and find minimum.
    List<float> list;
    float val;
    float minval = FLT_MAX;    // Max float value, from float.h
    while ( cin >> val) {
        if (val < minval) minval = val;
        list.add(val);
    }

    // Normalize values and write out.
    for (ListIterator<float> i(list); i.more(); i.advance()) {
        cout << i.current() - minval << endl;
    }

    return 0;
}
```

The first loop reads the values from cin, finds the minimum, and adds each value to the list; the second loop traverses the list, writing the normalized values to cout. The variable i is a ListIterator<float> object, an object used for traversing lists of float objects. This object is an example of an *iterator*, an object used for traversing a data structure. (Iterators are discussed in Section 13.7.) Once initialized from a list, a ListIterator<float> object is like a finger pointing at an item in the list. The member function advance() moves the finger to next item in the list, more() returns true if more items remain, and current() returns the item the finger is currently pointing at. In essence, the variable i is the analog of the integer loop index one would use to step along the elements of an array.

The List<T> class template implements the diagram shown earlier: Instances of the nested class Node hold a link and an item; a List<T> object holds a pointer to the first and last Node objects on the list, or null if the list is empty; the add() member creates a new Node and appends it onto the list, treating an empty list as a special case. The remove() member deletes any Node whose datum is equivalent to the member function's argument, as tested by == (see Exercise 6.9). ListIterator<T> is declared a friend so that it can access the nonpublic members of List<T>:

ch6/List.h

```
template<class T>
class List {
public:
    List() : first(0), last(0) {}

    void add(T x) {
        if (first == 0) first = last = new Node(x);
        else last = last->link = new Node(x);
    }

    void remove(T x);
    // ...

    friend class ListIterator<T>;
private:
    class Node {
    public:
        Node(T x) : link(0), datum(x) {}
        Node* link;
        T datum;
    };

    Node* first;
    Node* last;
};
```

Node is a class nested (cf. Section 4.6) in List<T>; it is private to List<T> because it is an implementation detail and is not necessary to use List<T>. The iterator object's finger is held in the member datum cur:

ch6/List.h

```
template<class T>
class ListIterator {
public:
    ListIterator(const List<T>& list) : cur(list.first) {}
    Boolean more()   const { return cur != 0;       }
    T       current() const { return cur->datum; }
    void    advance()       { cur = cur->link;    }
private:
    List<T>::Node* cur;
};
```

ListIterator<T> is a friend so that it can access the first member of List<T> and the link and datum members of Node.

List<T> and ListIterator<T> provide two different aspects of a single facility. Two classes are necessary to allow more than one simultaneous traversal of a list, just as several loops with separate loop control variables can traverse one array simultaneously. These two classes must be coupled tightly: ListIterator<T> exists to hide the details of the list implementation, allowing the user to traverse a list without knowing its details. The implementation of ListIterator<T> is part of the implementation of the overall list facility. The friend declaration expresses this fact.

Friend declarations introduce additional coupling between a class and one or more functions, and coupling makes program modifications more costly.

- Use friend declarations sparingly.

6.11 Input/Output Operators for Classes

As we discussed in Sections 2.4 and 3.5, the >> and << operators are used, in conjunction with the iostream class system, for input and output of built-in objects. Consistent facilities for input and output of class objects can be obtained by providing these operators for the class objects.

The ostream class, which is supplied as part of the iostream class system, is used for providing output operators that work with all output streams, whether connected to the terminal, to a file, or even to an in-memory buffer. It has an operator<<() member function for each built-in type. To supply an output operator for a class, write an << operator that takes an ostream as its left operand and a class object as its right operand. Here is the declaration for the output operator for the Interval class on page 157:

ch6/demoFal.C

```
extern ostream& operator<<(ostream& os, const Interval& i);
```

The first argument, which corresponds to the left operand, is a reference to the ostream to write to; the second argument, which corresponds to the right operand, is a const reference to the class object. By convention, the operator function returns a reference to the ostream object so that << operators can be chained together as with the built-in versions. Here is an implementation:

ch6/demoFal.C

```
ostream& operator<<(ostream& os, const Interval& i) {
    return os << '[' << i.lo() << ", " << i.hi() << ']';
}
```

A user-defined output operator is used just like an output operator for a built-in type:

ch6/demoFal.C

```
Interval small(-.01, .02);
// ...
cout << small << endl;
```

The istream class is the input analog of ostream. It is used when defining the >> operator for class objects:

ch6/demoFal.C

```
extern istream& operator>>(istream& is, Interval& i);
```

For input, the second argument must be a non-const reference to the class object because the class object is going to be modified. Assuming that the input format for an interval is two numbers separated by a comma and enclosed in square brackets (i.e., the same as the output format), the input operator looks like this:

ch6/demoFal.C

```
istream& operator>>(istream& is, Interval& i) {
    char lbracket = 0;
    char rbracket = 0;
    char comma = 0;
    double lo;
    double hi;
    is >> lbracket >> lo >> comma >> hi >> rbracket;
    if (lbracket != '[' || comma != ',' || rbracket != ']') is.clear(ios::badbit);
    else if (is) i = Interval(lo, hi);
    return is;
}
```

A state is associated with each stream (input or output). When an error occurs, the state of the stream is set to bad and future operations on that stream have no effect until the state is reset. The function call is.clear(ios::badbit) sets the state of stream is to bad; it can be reset by is.clear(). By convention, when an operation on an input stream fails, the destination object is not altered. Our >> operator begins by reading the expected inputs; if any read was not successful, the corresponding variable will not be set properly and we set the stream state to bad. The last if stores the values read into the destination object i if the stream is in the good state (a stream tests true when in the good state).

6.12 Notes and Comments

6.1 C++'s coupling of implicit conversions and single argument constructors results in surprising behavior more often than one would like. There is no mechanism for providing a single argument constructor without defining a user-defined conversion; such a mechanism would be useful. Similarly, there is no way to distinguish between use of a copy constructor for initialization and for function return.

6.2 We use the Boolean class throughout this book as a documentation aid. To C++, zero is false and nonzero is true: Numbers, usually of type int, represent logical values. Declaring a variable to have type Boolean indicates clearly that the variable holds logical values, not numbers to be used in a numeric computation.

However, this practice introduces a potential problem: Two independently written pieces of code that each use a different class called Boolean cannot be used together without reconciling the differences between the two Boolean classes. Of course, such conflicts are not unique to Boolean classes.

At present, if you are writing a large amount of software intended for use by others, it is wise to adopt a naming convention such as adding a unique prefix to all class and global function names. The ANSI C++ standardization committee is considering a namespace mechanism that, if adopted, should ameliorate this problem.

6.3 Portions of the Fallible<T> class of Section 6.4.4 are independent of the particular type T. The class template Fallible<T> can be factored into a type-independent (non-template) class and a type-dependent class template. See Section 11.2.1.

6.4 The statement

ch6/mult-assign.C

```
(a = b) = c;  // Assign b to a, then c to a; b is unchanged
```

is misleading but correct. Meyers and Lejter [83] recommend that assignment operator functions should return a const reference to their left operand. Following this advice is inconsistent with C++'s behavior for built-in objects, but it makes the foregoing statement illegal.

6.5 Problems introduced by implicit type conversions were elucidated in [84].

6.6 Input/output errors should be handled with exceptions. However, implementations of C++ exceptions were not available when the stream classes were designed, so they provide error handling via the states described in Section 6.11.

6.7 The use of enumerations in classes to create symbolic constants with class scope is described in [27, Section 2.7].

6.13 Exercises

6.1 Why would omitting the const version of Point::x() make it impossible to print the coordinates of p2 in the example on page 144?

6.2 Reimplement the SimpleArray<T> class template described in Section 4.3 using const member functions where appropriate.

6.3 Write a LineSegment class that allows the code in Section 6.2 to work.

6.4 Given a class C, the constructor C::C(C) is illegal. Explain why this must be so.

6.5 Design and implement a SimpleArray2d<T> class template that provides a two-dimensional analog of the SimpleArray<T> class template, as modified in Exercise 6.2. *Hint:* Use the function call operator for subscripting.

6.6 Add a conversion operator to the LineSegment class of Exercise 6.3 that converts to a Line from page 90. Add a constructor to the resulting class that takes a Line object. Why is this difficult?

6.7 Implement the elseDefaultTo() member of Fallible<T>.

6.8 Add an operator+=() to the Point class on page 87; add a global operator+() taking two Point objects that calls operator+=(). Test them.

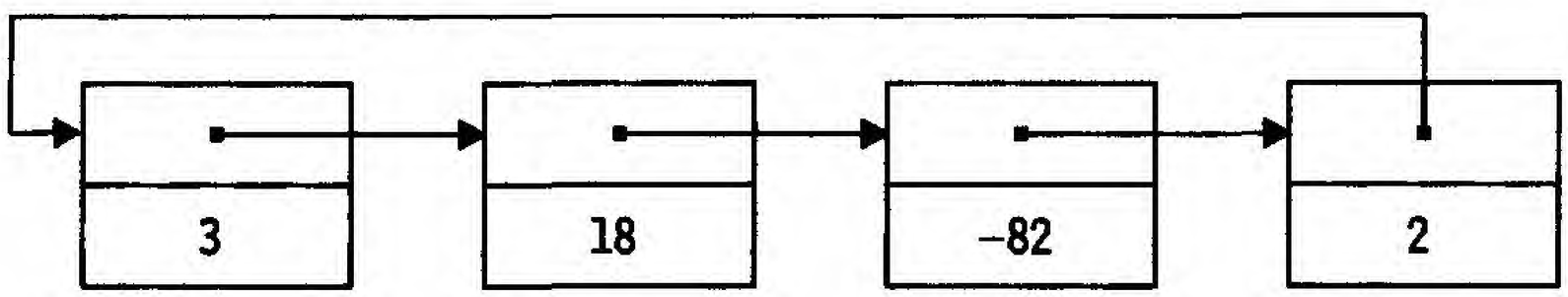
6.9 Write the member function List<T>::remove() for the list template on page 170. Test it.

6.10 Two essential member functions are not shown for the List<T> class template on page 170. What are they? Provide implementations.

6.11 All of the members of the nested class List<T>::Node (page 170) are public. Are they accessible from outside of List<T>? Why or why not?

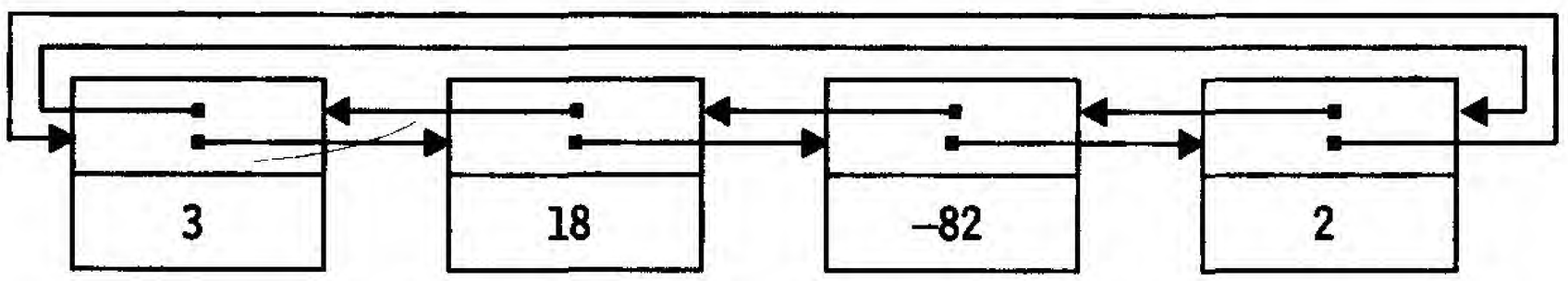
6.12 Revise the List<T> and ListIterator<T> class templates (page 170) so that the iterator is a nested class in the list class. Discuss the advantages and disadvantages of using class nesting in this situation.

6.13 Design and implement classes for a *circular linked list*, a linked list that has no beginning or end. For example, the linked list shown on page 168 would become



Include in your iterator class the ability to access the item following the current item.

6.14 Design and implement classes for a *circular doubly linked list*, a circular linked list that can be traversed in either direction. For example, the circular linked list shown for Exercise 6.13 would become



Include in your iterator class the ability to access the items preceding and following the current item.

6.15 Does the implementation of the >> operator for Interval on page 172 allow blanks to appear before or after the commas in the input? Why or why not?

CHAPTER 7

Object Lifetime and Memory Management

ONLY rarely are the sizes of all the data structures used in a computing problem known when the program is written, or even when it is compiled. How large will the matrices be? How many constraint equations are there in a linear programming problem? It is not adequate to set some huge maximum size for each data structure. One person's idea of huge will be tiny to another: Sooner or later programs with fixed-size limits fail. Moreover, adopting huge fixed-size data structures wastes resources, competing with other uses of memory.

C++ provides mechanisms for *memory management*, for controlling the use of memory during program execution. Besides being important for adapting programs to variable-size problems, memory management and the related topic of pointer classes are key to C++'s support for interfaces, an important concept introduced in Chapter 9. We discuss memory management in this chapter and pointer classes in Chapter 14. We begin with the life cycle of objects.

7.1 Object Life Cycle

An object's existence consists of seven stages:

Allocation. Object creation begins when C++ allocates storage for the object; the memory is said to be allocated to the object. *Allocation* is triggered by code that creates an object.

Preinitialization. Preinitialization sets up bookkeeping information that C++ uses to implement object behavior (e.g., the virtual functions described in Chapter 9). Allocation provides raw memory for an object; preinitialization endows it with the properties that cause it to behave like a C++ object.

Initialization. Initialization consists of recursive initialization of the object's data members and base subobjects (see Section 10.1), followed by the final step in object creation, execution of the body of the constructor that was invoked to

trigger creation of the object. Initialization of an object of built-in type sets the object's value.

Use. Once allocated and initialized, an object is used by calling its member functions or accessing its data members.

Cleanup. Cleanup is the inverse of initialization. Cleanup consists of execution of the body of the destructor member function, followed by recursive destruction of the object's data members and base subobjects. For example, initialization may cause a file to be opened; it might be necessary to close the file as part of cleanup. There is no cleanup for objects of a built-in type.

Post cleanup. This is the inverse of preinitialization, namely transforming an object back into raw storage.

Deallocation. The object's storage is made available for other use; the memory is said to be *deallocated* or *freed*.

Allocation, preinitialization, and initialization occur sequentially, triggered by the code that creates an object. These steps are collectively called *construction*. Allocation is either initiated by C++, when a local variable is declared, or by the programmer, using the new operator. Preinitialization is compiler dependent and cannot be controlled by the programmer. Initialization is controlled by the programmer using constructor member functions.

Cleanup, post cleanup, and deallocation also occur sequentially, triggered either explicitly by a delete operator or automatically by C++. These steps are collectively called *destruction*. Post cleanup is compiler dependent and cannot be controlled by the programmer; typically, an object's bookkeeping information will be forgotten and no operations occur during post cleanup in many compilers.

7.2 Object Lifetime

An object's *lifetime* is the time during which memory is allocated to it, the time between the allocation and deallocation of its memory. C++ objects have one of three lifetimes:

1. *Static* objects are allocated once and are not freed until the program terminates.
2. *Automatic* objects are allocated when their declarations are executed and freed automatically when the block containing them terminates.
3. *Dynamic* objects are allocated and freed in arbitrary order under the programmer's control.

To illustrate the various lifetimes of objects, we define a TraceLifetime class that prints a message each time an instance is created or destroyed:

ch7/TraceLifetime.h

```
class TraceLifetime {
public:
    TraceLifetime(const char* variable_name);          // Create and print message.
    ~TraceLifetime();                                  // Destroy and print message.
private:
    enum { max_objects_to_trace = 100 };               // Maximum number of
                                                       // objects to trace
    static char existing_objects[1+max_objects_to_trace];  // L = live; D = destroyed.
    static int total_created;
    int instance_number;                               // Creation sequence number.
    char var_name[21];                                 // Name of variable (first 20
                                                       // chars + null)
};
```

The constructor takes a character string to be incorporated in the messages. The static member total_created tracks how many instances have been created, and the static member existing_objects tracks how many instances have been created and then destroyed. Each time a new instance is created, a letter L (for Live) is appended to the string; each time an instance is destroyed, its letter L is changed to a letter D (for Destroyed). The implementation is straightforward:

ch7/TraceLifetime.C

```
int TraceLifetime::total_created = 0;
char TraceLifetime::existing_objects[1+TraceLifetime::max_objects_to_trace] = "";

TraceLifetime::TraceLifetime(const char* variable_name) :
    instance_number(total_created++) {
    // Copy first 20 characters of variable name
    strncpy(var_name, variable_name, 20);
    var_name[20] = '\0';

    if (total_created <= max_objects_to_trace) {
        existing_objects[total_created-1] = 'L';
        existing_objects[total_created] = '\0';
        cout << existing_objects << " TraceLifetime constructed for " <<
                                    var_name << "." << endl;
    }
}
```

```
TraceLifetime::~TraceLifetime() {
    if (instance_number < max_objects_to_trace) {
        existing_objects[instance_number] = 'D';
        cout << existing_objects << " TraceLifetime destroyed for " <<
                                        var_name << "." << endl;
    }
}
```

The strncpy() function is declared in the ANSI C header string.h; it is used here to copy characters from the string pointed to by variable_name into the string pointed to by var_name, stopping if variable_name has more than 20 characters.

In the next three sections, we use TraceLifetime to illustrate the lifetime behavior of static, automatic, and dynamic objects.

7.3 Static Objects

Static objects are allocated once and are not freed until the program terminates. Once allocated, the lifetime of a static object extends for the duration of the program's execution.

All static objects are associated with a variable: The only way to create a static object is to define a variable. Therefore the allocation of static objects depends on the scope of their associated variables. Variable names can have *class scope* (declared as a member of a class), *file scope* (declared outside all blocks and classes), or *local scope* (declared in a block).

To maintain C compatibility, C++ defines a split idea of scope for file scope names. A file scope name can have either *internal linkage* (known only within the translation unit in which it is declared) or *external linkage* (known throughout the program). Thus static objects come in four combinations: class scope, file scope with external linkage, file scope with internal linkage, and local scope. We discuss these four cases in the next subsection. Of these four, class scope static objects (i.e., static data members) are the most useful. We treat their construction and destruction in Section 7.3.2.

7.3.1 Scope and Linkage

To avoid cumbersome phrases like "the object associated with a name having file scope and external linkage," we'll speak of the scope and linkage of objects; bear in mind the more precise meaning.

Class Scope An object declared static inside a class (i.e., a static data member) has class scope, static lifetime, and external linkage. Static data members were discussed in Section 4.1.7.

File Scope with External Linkage An object declared outside all blocks and classes and without the static keyword has file scope and external linkage. Every declaration of an object with external linkage, even in different translation units, refers to the same object. File scope objects with external linkage are like FORTRAN COMMON blocks: Any part of the program can access or modify the object. Every declaration of an object with external linkage must be identical. Furthermore, there must be exactly one definition in the entire program for each such object.

The desire for compatibility with C has resulted in complicated rules for determining when a name has internal or external linkage and when a file scope declaration is also a definition. To avoid confusion, ARM §3

- Use the extern keyword to declare file scope names with external linkage; omit the extern keyword on their definitions.

For example,

ch7/static-linkage.C

```
extern int a;          // Declaration, external linkage
extern Complex c1;     // Declaration, external linkage
```

Every file that uses these variables should include these declarations. In one file, the corresponding definition should appear:

ch7/static-linkage.C

```
int a = 1;          // Definition, initialized to 1;
Complex c1;         // Definition, default constructor.
```

File scope objects with external linkage can lead to accidental name clashes:

- Avoid file scope objects with external linkage; use class scope instead.

For example, using file scope names with external linkage, you might put the following declarations in a header file:

ch7/class-scope.h

```
extern const double c;      // Speed of light
extern const double G;      // Gravitational constant
extern const double k;      // Boltzmann's constant
// ...
```

The corresponding definitions would appear in one .C file:

ch7/class-scope.C

```
const double c = 3.00e8;          // Speed of light
const double G = 6.67e-11;        // Gravitational constant
const double k = 1.38e-23;        // Boltzmann's constant
// ...
```

It is not difficult to imagine conflicts with these names. Using class scope, we could put the following class definition in a header file:

ch7/class-scope.h

```
class PhysicalConstants {
public:
    static const double c;     // Speed of light
    static const double G;     // Gravitational constant
    static const double k;     // Boltzmann's constant
    // ...
};
```

The corresponding initializations would appear in one .C file:

ch7/class-scope.C

```
const double PhysicalConstants::c = 3.00e8;        // Speed of light
const double PhysicalConstants::G = 6.67e-11;      // Gravitational constant
const double PhysicalConstants::k = 1.38e-23;      // Boltzmann's constant
// ...
```

Name conflicts with PhysicalConstants are still possible but are less likely than with names like c, G, or k.

File Scope with Internal Linkage A file scope object declared with the static keyword has static lifetime and internal linkage. File scope objects with internal linkage treat a file as a unit of programming: All functions in the file following the object declaration have access to the object, but no other functions know about it. Class scope static objects provide the same scope control and better reflect the organization of our programs.

- Prefer class scope names to file scope names.

Local Scope A local variable declared with the keyword static has local scope and static lifetime. Such static objects are initialized the first time that their definition is executed. If the definition is never executed, the object is never created. Local scope static data put state into a function, contradicting the natural assumption that the result of calling a function is independent of prior calls to the same function. Use class objects to couple state and behavior:

- Replace functions that hold state in static local variables with a member function and member data.

Static local variables can sometimes be used to improve the performance of a function without altering its behavior. Suppose that a cpu-intensive function without side-effects might be called repeatedly with the same argument. Saving the previous result (called *caching* it) could be worthwhile:

ch7/cache.C

```
extern float f(float);    // Expensive computation

float remembers(float x) {
    static int first_time = 1;
    static float previous_x;
    static float previous_result;

    if (first_time || x != previous_x) {
        first_time = 0;
        previous_x = x;
        previous_result = f(x);
    }
    return previous_result;
};
```

The expensive computation is done the first time remembers() is called and anytime the argument has a different value than previous_x. Even this use of static data is done better with a class; see Exercise 17.3.

7.3.2 Construction and Destruction of Static Objects

Static objects are constructed once and destroyed when the program terminates. Destructors called during program termination can perform various cleanup operations. The destructor for an object that represents a physical object might power-down the physical object or cause it to return to a quiescent state, such as a robot moving to a home position. Or consider an object that periodically records experimental measurements but only writes them to disk when an internal buffer fills; the destructor for such an object could write out any measurements that remain in the buffer.

We saw earlier that static objects with local scope are constructed when their definition is executed. We now also consider the sequence in which nonlocal static objects (either file or class scope) are constructed and destroyed.

Order within a Translation Unit The following code, contained in a single file, uses TraceLifetime (page 177) to illustrate the construction and destruction of static objects:

ch7/tStaticNesting.C

```
static TraceLifetime s1("s1");                    // File scope static object.

void fnc(int j) {
    cout << "-- starting fnc(" << j << ") --" << endl;
```

```
    static TraceLifetime s2("s2");          // Local scope: created on first call to fnc().
    if (j == 2) {
        static TraceLifetime s4("s4");      // Only created when j == 2.
    }
    for (int i = 1; i <= 2; i++) {
        cout << "-- loop i=" << i << " --" << endl;
        static TraceLifetime s3("s3");      // Local scope: created once, when i == 1.
    }
    cout << "-- returning from fnc(" << j << ") --" << endl;
}

int main() {
    cout << "-- main starts --" << endl;
    fnc(1);
    fnc(2);
    cout << "-- main ends --" << endl;
    return 0;
}

static TraceLifetime s5("s5");              // File scope static object.
```

Running this code produces the output shown in Fig. 7.1.

I §3.4 Nonlocal static objects in a translation unit are constructed before the first use of any function or object defined in that translation unit. Therefore s1 and s5 are initialized before main() executes, despite the positions of their declarations in the source file. Within a translation unit, nonlocal static objects are constructed in the
12.6.1 order in which they appear. Thus the compiler must construct s1 before s5.

Static local scope objects are constructed the first time that their definition is executed (or not at all if the definition is not executed) and are destroyed when main() terminates. Thus s2 and s3 are constructed during the first execution of fnc(); s4 is constructed during the second execution of fnc(), which is the first time it is invoked with an argument of 2. Notice that the order in which the definitions appear in the file is irrelevant.

Once allocated, a static object remains until the program terminates by returning from main() or by calling exit(). Each static object is destroyed (and its destructor member function run) as part of terminating the program. If the program terminates by calling abort(), or the operating system forces the program to terminate, the destructors are not called. Static objects with nonlocal scope are destroyed in the inverse order of their construction; local scope static objects are destroyed in

```
L TraceLifetime constructed for s1.
LL TraceLifetime constructed for s5.
-- main starts --
-- starting fnc(1) --
LLL TraceLifetime constructed for s2.
-- loop i=1 --
LLLL TraceLifetime constructed for s3.
-- loop i=2 --
-- returning from fnc(1) --
-- starting fnc(2) --
LLLLL TraceLifetime constructed for s4.
-- loop i=1 --
-- loop i=2 --
-- returning from fnc(2) --
-- main ends --
LDLLL TraceLifetime destroyed for s5.
LDLDL TraceLifetime destroyed for s3.
LDLDD TraceLifetime destroyed for s4.
LDDDD TraceLifetime destroyed for s2.
DDDDD TraceLifetime destroyed for s1.
```

Figure 7.1 Program Output Illustrating the Construction and Destruction of Static Objects

arbitrary order, both with respect to each other and with respect to nonlocal scope objects.

Order across Translation Units Nonlocal static objects defined in the same file are constructed in order of their appearance; no other ordering of their construction is dictated. In particular, a nonlocal static object defined in one file is not guaranteed to be initialized before it is used in another file. For example, suppose we have an arbitrary precision rational number class

ch7/Rational.h

```
class Rational {
public:
    Rational(int numerator, int denominator);
    // ...
    static const Rational zero;
    static const Rational quarter;
    static const Rational half;
    // ...
};
```

with some common constants defined like this:

ch7/Rational.C

```
const Rational Rational::zero(0, 1);
const Rational Rational::quarter(1, 4);
const Rational Rational::half(1, 2);
```

We might then define a RationalComplex class

ch7/RationalComplex.h

```
class RationalComplex {
public:
    RationalComplex(const Rational& real, const Rational& imag);
    // ...
    static const RationalComplex realHalf;
    static const RationalComplex imagHalf;
    // ...
};
```

with its own constants:

ch7/RationalComplex.C

```
const RationalComplex RationalComplex::realHalf(Rational::half, Rational::zero);
const RationalComplex RationalComplex::imagHalf(Rational::zero, Rational::half);
```

However, the constructor for RationalComplex::realHalf might run before the constructor for Rational::zero, giving unpredictable results.

- Don't initialize a static object defined in one file with a static object defined in another file.

In simple situations like this one, you can put all of the static member definitions in one file, thus getting control over the initialization order. This solution is cumbersome in more complicated examples and not possible for class libraries; see Notes and Comments 7.3.

7.3.3 Default Initialization to Zero

A variable of built-in type is considered uninitialized if you don't supply an initial value in the variable's declaration, either enclosed in parentheses or after an equals sign. In cases in which such a variable is static, the compiler will initialize the variable to zero before any constructor functions are run for any variables. For example, the following code works as indicated by the comments:

ch7/zero.C

```
#include <complex.h>

extern float zero;          // Forward declaration

complex one(zero+1);        // Has value 1.0+0.0i
complex two(zero+2);        // Has value 2.0+0.0i
float zero;                 // Has value 0 by default.
```

The variable zero is defined but not initialized. Therefore the compiler initializes it to 0, converted appropriately to type float. This is done before any constructors are run. For this reason, the code works correctly despite the definition of zero appearing after the definitions of one and two.

It is occasionally useful to depend on a static object being initialized to zero before any constructors are run. See, for example, Exercise 7.3.

7.4 Automatic Objects

A local variable contained in a block but not declared static is an *automatic object*: It is constructed each time the variable definition is executed and destroyed when the containing block terminates, or, if in a loop, at the bottom of the loop body; if blocks are nested, the allocation and freeing of local variables is nested accordingly. ARM §6

The following program illustrates these behaviors:

ch7/tAutoNesting.C

```
int main() {
    cout << "-- main starts --" << endl;
    TraceLifetime a1("a1");
    for (int i = 1; i <= 2; i++) {
        cout << " -- loop i= " << i << " -- " << endl;
        TraceLifetime a2("a2");          // Object created each loop iteration.
        if (i == 2) {
            TraceLifetime a3("a3");      // Object created iff i == 2.
        }
    }
    cout << "-- loop is done -- " << endl;
    TraceLifetime a4("a4");
    cout << "-- main ends --" << endl;
    return 0;
};
```

The program produces the output shown in Fig. 7.2. The object a1 is created first in the block defined by main(). On the first pass through the for loop, a2 is created and then destroyed; on the second pass a2 is created again, but this time a3 is created and destroyed inside the if statement before a2 is destroyed. After the loop, a4 is created. At the end of main(), both a1 and a4 are destroyed. Automatic objects are often said to be "on the stack," after the name of the implementation technique that most compilers use for automatic objects.

When an exception is thrown, all of the active blocks between the throw and the corresponding catch are terminated and the destructors are run for all automatic objects that have been constructed in those blocks. For example, in the

```
-- main starts --
L TraceLifetime constructed for a1.
-- loop i= 1 --
LL TraceLifetime constructed for a2.
LD TraceLifetime destroyed for a2.
-- loop i= 2 --
LDL TraceLifetime constructed for a2.
LDLL TraceLifetime constructed for a3.
LDLD TraceLifetime destroyed for a3.
LDDD TraceLifetime destroyed for a2.
-- loop is done --
LDDDL TraceLifetime constructed for a4.
-- main ends --
LDDDD TraceLifetime destroyed for a4.
DDDDD TraceLifetime destroyed for a1.
```

Figure 7.2 Program Output Illustrating the Construction and Destruction of Automatic Objects

code that follows the destructors for c and then for a are run when f2() throws an exception:

ch7/tAutoException.c

```
void f2() {
    cout << "-- f2 starts -- " << endl;
    TraceLifetime c("c");
    throw "exception";
    cout << "-- f2 ends -- " << endl;
}

void f1() {
    cout << "-- f1 starts -- " << endl;
    TraceLifetime a("a");
    f2();
    TraceLifetime b("b");
    cout << "-- f1 ends -- " << endl;
}

int main() {
    cout << "-- main starts --" << endl;
    try {
        f1();
    }
```

```
-- main starts --
-- f1 starts --
L TraceLifetime constructed for a.
-- f2 starts --
LL TraceLifetime constructed for c.
LD TraceLifetime destroyed for c.
DD TraceLifetime destroyed for a.
Exception caught.
-- main ends --
```

Figure 7.3 Program Output Illustrating the Destruction of Automatic Objects when an Exception Is Thrown

```
    catch(const char*) {
        cout << "Exception caught." << endl;
    }
    cout << "-- main ends --" << endl;
    return 0;
}
```

Running this program produces the output shown in Figure 7.3. Notice that the destructor for `b` is not run because `b` was never constructed.

The automatic lifetime control provided by automatic variables can be used to acquire and release limited resources. For example, most operating systems limit the number of files that a program can process simultaneously (i.e., the number of open files). An open file is therefore a limited resource. A program that processes many files must ensure that each file that is opened is closed when the data in the file are processed. In the C++ I/O class library, an input file is represented by an `ifstream` object and an output file by an `ofstream` object. Their constructors take a file name and open the specified file; their destructors close the open file. A function that processes a file of experimental data might look like this:

ch7/resource-acq.C

```
void process_file(const char* in_name, const char* out_name) {
    ifstream in_file(in_name);
    ofstream out_file(out_name);

    // Process input file, generating output file
    // ...
    if (data_is_bad) throw BadData();
    // ...
}
```

The input and output files are closed when the corresponding ifstream and ofstream objects are destroyed. This occurs when the function terminates, whether normally or because an exception is thrown. The programmer can't forget to close one of the files.

7.5 Dynamic Objects

Dynamic objects are allocated and freed explicitly by the programmer. There is no *a priori* relationship between the lifetime of two dynamic objects, although a program's logic sometimes dictates an ordering among the lifetimes of dynamic objects. Dynamic objects are often said to be "in the heap" or "in the free store:" Most implementations allocate dynamic objects in a specific area of memory called the *heap* or *free store*.

The number of dynamic objects allocated is determined while the program is running, not when the program is written and compiled. This allows a program to allocate memory to objects when it knows how big the problem is and how to split the available memory (or address space) among its objects.

Two mechanisms allow the number of objects to be determined as the program runs. First, arrays of objects can be allocated dynamically, with the number of objects in the array determined when the array is created. Second, a dynamic object need not be associated with a variable: Its lifetime is not coupled *a priori* to the lifetime of a block or function. Contrast this with all FORTRAN-77 objects and with static and automatic objects in C++: These are always associated with a variable or are temporary objects constructed and destroyed by C++ in step with block lifetimes. Since the number of variables is fixed when a program is compiled, there can be only a fixed number of static and automatic objects. The decoupling of variables and dynamic objects permits a program to create as many dynamic objects as it requires, limited only by the total amount of memory available.

New freedom brings new responsibility for the programmer. The programmer explicitly allocates and frees dynamic objects. An allocation must be paired with one and only one deallocation. Deallocating twice almost always causes your program to crash; never deallocating can cause your program to use more and more memory as it runs. For programs that are not expected to run for a long time, this might be acceptable; you'd probably be unhappy, though, if your data-analysis program crashed after a few hours of use because it ran out of memory.

7.5.1 Construction

Construction of a dynamic object is initiated by the new operator and destruc-
3.3.3 tion by the delete operator. In its simplest form, the new operator takes a type name (built-in or class) as its operand, like this:

ch7/newDemo.C

```
int* p = new int;
```

The new operator allocates memory for the object and, for class objects, invokes the default constructor; new returns a pointer to the constructed object, or 0 if it was unable to allocate sufficient memory. If an argument list in parentheses is given following the type name, the arguments are used to select and call the type's constructor, or, for built-in types, to initialize the object. Thus

ch7/newDemo.C

```
int* q = new int(3);
```

allocates an int and initializes it to 3 and

ch7/newDemo.C

```
Rational* r = new Rational(8, 9);
```

allocates a Rational and invokes its (two argument) constructor.

An array of objects is allocated dynamically by providing array dimensions enclosed in square brackets:

ch7/newDemo.C

```
// Create array of int's with size determined at runtime
int num_elements;
cin >> num_elements;
int* qa = new int[num_elements];
```

Initialization arguments cannot be supplied with the array form, so arrays of class objects can only be created for classes with a default constructor.

- Provide a default constructor for every class possible.

For example, one can't make an array of rational numbers represented by the Rational class on page 183.

Dynamic objects add flexibility at the expense of space and time. We can only refer to dynamic objects via pointers initialized by the new operator. These pointers occupy space beyond the dynamic objects themselves, and many compilers impose one or two additional words of storage overhead per dynamic object. The running time of operator new is compiler dependent and can also depend on the program's history of memory allocation and deallocation. Some control over this overhead can be gained by providing a class-specific new operator; an example appears in [74, Section 6.3]. ARM §12.5

7.5.2 Destruction

Once allocated, destruction of a dynamic object is initiated by delete:

ch7/newDemo.C

```
delete p;
delete q;
delete r;
```

The delete operator takes a pointer that was returned by new, calls the destructor (if any) for the object the pointer points at, and then returns the object's memory to
3.4 the pool of memory available to be allocated by new. Conceptually (and sometimes actually), deleting an object alters the object. Therefore you can't delete a pointer to a const object. Invoking delete does not necessarily alter the pointer itself, so don't depend on delete setting the pointer to null. Deleting a null pointer is legal and harmless.

Arrays, which are allocated with new [ ], must be deleted with delete [ ]:

ch7/newDemo.¶

```
delete [ ] qa;
```

Bad things can happen if you use the delete without the square brackets on a pointer to an array, or if you use the square brackets form on a pointer that was not obtained from the square brackets form of new.

The fundamental rule for dynamic allocation and deletion of objects is simple:

- Match every invocation of new with exactly one invocation of delete of the same kind.

In practice, failure to follow this seemingly innocuous rule is the source of pernicious bugs. The worst problems arise from bad deletions (i.e., from deleting an object more than once, deleting a pointer not obtained from new, or using the wrong form of delete). As Ellis and Stroustrup [44, p. 63] say, "Bad deletions are usually not detected immediately, and programs containing them are therefore among the nastiest to debug. Almost any effort to avoid such bad deletions is worthwhile."

The simplest and often the best approach to avoid bad deletions is to write the code in one place and reuse it. We have already seen an example of doing this in the array classes of Sections 4.2 and 4.3. Dynamic allocation of the array elements is done by the array class's constructor; delete is invoked by the array class's destructor. The compiler arranges for the array's destructor to be invoked when the array is to be destroyed; we don't have to code the delete ourselves or decide when to invoke it. For example, if the array object is an automatic variable, its destructor is run whenever the containing block is terminated, even if by an exception (cf. page 187).

- Use array classes that incorporate management of dynamic arrays instead of using built-in dynamic arrays.

7.6 Preventing Dangling References and Garbage

Pairing new in a constructor and delete in a destructor is not sufficient to avoid bad deletions. Bad deletions can occur because of dangling reference created by

copying class members improperly, by pointing to deleted automatic objects, or by functions that delete their arguments. Too few deletions can result in excessive memory use and, ultimately, in running out of memory. We cover each of these cases in the following subsections.

7.6.1 Dangling Class Members

Consider the following class:

ch7/Dangle.C

```
class Dangle {    // WARNING: Incorrect code
public:
    Dangle()    { p = new int(0); }
    ~Dangle() { delete p;          }
private:
    int* p;
};
```

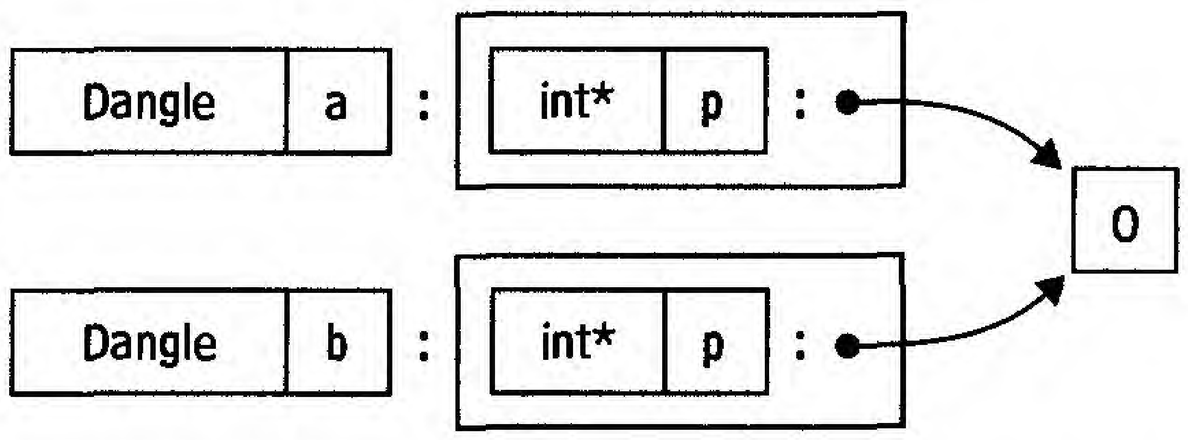
The constructor allocates an object with new and deletes the pointer with the correct form of delete. Yet all is not well.

Suppose we run the following code:

ch7/Dangle.C

```
void f() {
    Dangle a;
    Dangle b = a;
}
```

The second statement runs the C++-generated copy constructor to initialize b from a. C++-generated copy constructors copy the data members, so in this case b's pointer p is initialized with a copy of a's pointer p. Thus after the second statement, the objects look like this: **ARM**



When f() terminates, the destructor is run for both a and b; in each case, the pointer p is deleted, deleting one object twice, probably with disastrous consequences. Assignment would set up the same situation.

For most classes, we can solve this problem by providing our own copy constructor and assignment operator, like this:

ch7/Dangle.C

```
class NoDangle {
public:
    NoDangle()    { p = new int(0); }
    ~NoDangle() { delete p;         }
    NoDangle(const NoDangle& x) : p(new int(*x.p)) {}
    NoDangle& operator=(const NoDangle& rhs) {
        if (rhs.p != p) {
            delete p;
            p = new int(*rhs.p);
        }
        return *this;
    }
private:
    int* p;
};
```

The copy constructor allocates a new int object and initializes it to the value of the int pointed to by the Dangle object being copied. The assignment operator is similar, except that it must check for assigning an object to itself and it must delete the int object pointed to by the left operand. We saw a practical application of this solution, motivated by logical correctness, in Sections 6.3 and 6.6.

- Provide a copy constructor and assignment operator for classes that have a built-in pointer data member that is deleted by the destructor.

7.6.2 Dangling Pointers to Automatic Objects

The preceding example of deleting an object twice is a specific case of a more general problem. A pointer's lifetime is independent of the lifetime of the object to which it refers. If pointer p points to object b, any use of p after the lifetime of b is a *dangling reference*. Lest you think that dangling references are a problem just with classes that use dynamic objects, here is an example with neither classes nor dynamic objects:

ch7/dangle.C

```
void dangle(int j) {    // WARNING: Incorrect code
    int* p;
    if (j < 100) {
        // Dangling reference created when this block terminates
        int iarray[100];
        p = iarray;
    }
    else p = new int[j];
    for (int i = 0; i < j; i++) p[i] = i;
}
```

For values of j less than 100, a dangling reference is created when the if block terminates. The pointer p is left pointing to objects that no longer exist because C++ deletes automatic objects regardless of whether there are any pointers or references to them.

- Avoid pointers to automatic objects.

7.6.3 Pointers Left Dangling by Function Calls

The dangling reference problem can also arise from improperly deleting a dynamic object:

ch7/dangle.C

```
void f(int* x) {
    // ...
    delete [ ] x;                    // Causes dangling reference
}

void dynamic_dangle(int size) {      // WARNING: Incorrect code
    int* iarray = new int [size];
    f(iarray);
    for (int i = 0; i < size; i++) cout << iarray[i];
}
```

The function dynamic_dangle() allocates an array and passes it to f(), which deletes it. Further uses of the array are incorrect. Functions that delete their arguments can result in dangling references that propagate through many levels of function calls.

- Don't delete a function argument.

Dangling reference bugs are often difficult to find because the pointer continues to point to the deleted object's memory. It is not until that memory is used for some other purpose that the dangling reference manifests itself. That can happen in any part of the program, at any time. When or even if it happens might depend on input data. We have known people who spent weeks tracking down a dangling reference bug in a large program.

7.6.4 Garbage Memory

The converse of the dangling reference is the *garbage problem*, in which we fail to delete a heap object that is not pointed to by any accessible pointer. Such an object is called *garbage*. With infinite memory, we could ignore garbage: As a program ran it would consume more and more memory, never bothering to pick up after itself. For some small programs, the infinite memory assumption is essentially valid and garbage can be ignored. But many programs must reclaim and reuse memory. The garbage problem—failing to delete—is the opposite of

the dangling reference problem—deleting too soon. Our advice on handling the garbage problem is much the same:

- Hold pointers in class objects and pair new and delete in the class's constructors and destructors.

Using pointers requires care. Again, getting the code right once and reusing it is often the best way to avoid problems. In Chapter 14, we will develop pointer classes that can be used to handle many memory management details automatically. See also Notes and Comments 7.5.

7.7 Notes and Comments

7.1 The term *block scope* more precisely describes the scope referred to in C++ descriptions as local scope. Local is a relative term: A declaration in a class is local to the class but it does not have local scope. Similarly, *global scope* means *file scope* with unspecified linkage, but it is commonly used to mean objects with external linkage. Such imprecise or confusing usage of terms is part of the cost of C compatibility.

7.2 Short file scope names with external linkage are likely to create conflicts at link time. This goes by the colorful term *name-space pollution.*

7.3 Authors of a class library often need to be sure that the library's static data members are initialized before a client uses them. Putting the static object definitions in the client's code is not viable in a library intended for general use or for which source code is not available to the client. One alternative is to make the static data member private and access it through a member function: Put the member function definition in the same file as the static data member. Since nonlocal static objects in a translation unit are initialized before functions in the same translation unit are used, the data member will be initialized before the client accesses it through the member function.

Where direct access is required, this solution is not available. Some compilers supply a compiler-specific mechanism for controlling the order in which static objects are initialized. Not being portable, this is to be avoided. Another alternative is to allow the static objects to be initialized by their default constructors and then arrange to assign the appropriate value before a client can access the static object. This technique applied to the Rational class looks like this:

ch7/Rational1

```
class Rational {
public:
    Rational();
    Rational(int numerator, int denominator);
    // ...
    static Rational zero;
    static Rational quarter;
    static Rational half;
    // ...
};
```

```
class RationalInitializer {
public:
    RationalInitializer();
private:
    static int count;
};

static RationalInitializer riInit;
```

As usual, to use the Rational class, the implementation of RationalComplex (and every other file that uses Rational) includes the header file that defines Rational, Rational1.h. Each such file then has a nonlocal static object riInit and that object is initialized before any use of Rational. The first time the constructor for RationalInitializer is executed, it assigns the appropriate value to each static member of Rational:

ch7/Rational1.C

```
RationalInitializer::RationalInitializer() {
    if (count++ == 0) {
        Rational::zero = Rational(0, 1);
        Rational::quarter = Rational(1, 4);
        Rational::half = Rational(1, 2);
    }
}

int RationalInitializer::count = 0;
```

Cleanup actions can be implemented by a destructor that decrements the count, doing the action at zero.

7.4 The approach to acquiring and releasing limited resources described in Section 7.4, sometimes called *resource acquisition is initialization,* is discussed in more detail in [107, Section 9.4].

7.5 LISP and Smalltalk systems use automatic garbage collection for memory management: Garbage collection is the process of finding all garbage and making it available for use. Clever techniques have been invented to minimize the runtime overhead of garbage collection in LISP and Smalltalk [113]. Some consideration has been given to adding garbage collection to C++, but substantial problems must be overcome with present implementations [51, Section 14.4.2]. An approach to garbage collection for C++ in which the garbage collector runs in parallel with the C++ program is described in [33]; another approach is described in [32]. Problems that arise when a language has both destructors and garbage collection have been considered in [7, 8]. A survey of classical garbage collection techniques appears in [24]. Most texts on the design and implementation of programming languages cover the interaction between storage management and other aspects of programming languages (see, e.g., [92, Chapter 8]).

7.6 Section 7.3.3 described the default initialization to zero of static objects that are not initialized explicitly. For built-in types, the meaning is clear. For class objects, the meaning is a bit subtle. What does initialization to zero mean for a class object? One would

ARM §

expect it to mean whatever calling the class's constructor with a zero argument means. But in this context it means something quite different: Consider the class object to be an aggregate of built-in objects (i.e., recursively decompose all data members into their constituent built-in types, then set all of those built-in objects to zero). Do this for our Rational number class, and you get the ill-defined number $\frac{0}{0}$, not the number $\frac{0}{1}$.

We recommend that you avoid depending on default initialization of static class objects to zero.

7.8 Exercises

7.1 What does the following code produce? (Guess before you try it!)

ch7/mixedNesting.C

```
TraceLifetime s1("s1");

int main() {
    static TraceLifetime s2("s2");
    TraceLifetime a1("a1");
    TraceLifetime a2("a2");
    for (int i = 1; i <= 2; i++) {
        TraceLifetime a3("a3");
        static TraceLifetime s3("s3");
    }
    TraceLifetime a4("a4");

    return 0;
}
```

7.2 Complete the definition of Rational and RationalComplex (page 183) sufficiently to experiment with variations in the order of statements and the order in which files are given to the linker. What effects do these variations have on your system? Guess the rule that your system chooses for ordering static constructors across translation units.

7.3 (Suggested by V. Milenkovic, Harvard University.) Suppose that you are part of a team of 10 people writing an interactive program. The other nine people are writing subsystems that implement various commands; you are responsible for the overall interaction with the program's user, including constructing a menu of commands and invoking commands as they are selected. Since the command set changes frequently, you want to avoid having to recompile your part of the program whenever a command is added or deleted. Write a Command class so that a subsystem can provide a command simply by creating a static Command object. Each Command object should be initialized with a const char*, giving the command name and a pointer to a function with no arguments to be executed. Hint: Use static constructors (constructors for objects with static lifetime) to build a linked list of commands, using an uninitialized static pointer object to start off the list.

7.4 Modify your solution to Exercise 7.3 to allow a subsystem temporarily to add subcommands to the list of commands. For example, when activated, an edit subsystem might

add change, add, delete, and quit subcommands; when deactivated, the edit subsystem should remove those commands from the list. Your solution should remove the subcommands added by a subsystem even if the subsystem terminates by throwing an exception.

7.5 Guess what is wrong with the following program and try to fix it:

ch7/tDynLifetime.(

```
int main() {
    TraceLifetime* p = new TraceLifetime[5];
    return 0;
}
```

7.6 Rewrite dangle() from page 192 using SimpleArray<int> from page 101. Why is a dangling pointer difficult to create when the array class replaces the built-in array?

7.7 Recode the dynamic_dangle() example from page 193 using SimpleArray<int>. Recode the original version with f() declared to take a const int* argument rather than an int*. What happens?

7.8 Apply the technique of Notes and Comments 7.3 to the code in Exercise 7.2 to initialize the Rational constants before the RationalComplex ones.

CHAPTER 8

An Example Program

In the preceding chapters, we have taken a quick tour of C++'s most basic concepts and features, focusing more on individual language features than on how they can be used together to solve a programming problem. In this chapter, we take the opposite tack: We start with a programming problem and tackle it with C++. To prepare you for the more sophisticated uses of C++ to come in subsequent chapters, this chapter reinforces and amplifies the material of the preceding chapters and introduces the importance of abstraction, information hiding, and encapsulation in C++ programming.

Section 8.1 introduces the problem and Section 8.2 solves it in a conventional way. At the end of that section, we critique the conventional solution. Section 8.3 introduces abstraction and encapsulation as design principles. Section 8.4 applies these principles to the sample problem. Section 8.5 refines this solution by applying information hiding, a design principle that fosters modifiability.

8.1 The Problem: Representing a Mesh

The problem we shall solve arises in the numerical solution of partial differential equations by the finite element method. The example itself is simple and doesn't require any background in partial differential equations or the finite element method. If you are not interested in the practical motivation, look at Fig. 8.1 and at Fig. 8.3 and skip the next paragraph.

The finite element method is commonly used for solving structural analysis problems and is also becoming popular in other areas of engineering. The phenomenon of interest is assumed to be governed by a partial differential equation over a domain Ω of the form

$$a\frac{\partial^2 u}{\partial x^2} + 2b\frac{\partial^2 u}{\partial x \partial y} + c\frac{\partial^2 u}{\partial y^2} + d\frac{\partial u}{\partial x} + e\frac{\partial u}{\partial y} + fu + g = 0,$$

where x and y are coordinates for Ω, $u(x, y)$ is the unknown function to be determined, and a, b, c, d, e, f, and g are known functions of x and y. The finite element

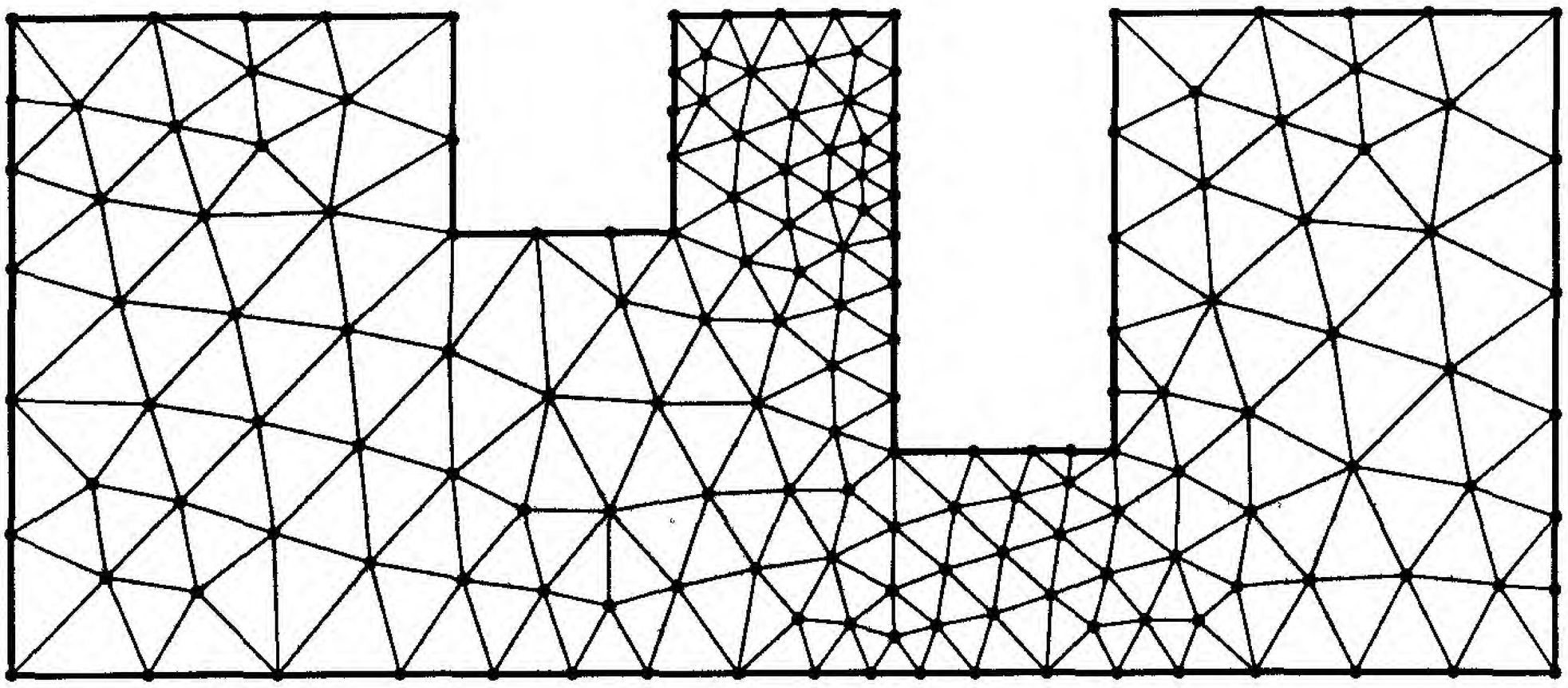


Figure 8.1 Example Mesh. The domain boundary is shown in boldface. All elements are triangles, and nodes are shown as solid dots. (This mesh was created by the algorithm described in [103].)

method is a general technique for constructing approximations to the function $u(x, y)$. The domain Ω is represented by the union of a finite number of subdomains Ω_j, called *elements*. Each element is a simple geometric shape, such as a triangle or a quadrilateral; and on each, a small number of points, called *nodes*, are identified. Given the equation to be solved, the preceding discretization of Ω into nodes and elements, and appropriate boundary conditions, a system of algebraic equations for the values of u at the nodes is constructed and solved. Once u is known at the nodes, its value anywhere in Ω can be obtained by interpolation. Figure 8.1 illustrates a typical two-dimensional domain discretized into nodes and triangular elements. A discretization of a domain is commonly referred to as a *mesh*.

The overall problem we tackle is to develop a mesh representation (i.e., data structure defining a mesh) that can be used by the various parts of a finite element analysis program. Even a toy finite element analysis program would be too complex to develop here. We focus on two tasks: writing an input module and checking mesh quality. The input module of a finite element program reads an input file that defines the mesh. A practical input module would also read physical parameters, loading information, boundary conditions, and a formulation of the equation to be solved. We shall ignore these complexities. Checking the mesh quality gives a rudimentary evaluation of how the representation can be used to solve finite element problems.

The input consists of

```
8          5          2.0 6.0    2.0 2.0    4.0 4.0    4.0 1.0    6.0 5.0
6.0 2.0    8.0 3.0    9.0 1.0    3          1 2 0      3          3 2 1
3          5 2 3      4          6 4 2 5    3          7 6 5
```

Figure 8.2 Sample Input Data that Defines the Mesh Shown in Fig. 8.3. The input is sequenced from left to right, top to bottom.

1. The number of nodes n_n;
2. The number of elements n_e;
3. The n_n node coordinates (pairs); and
4. The n_e elements, each specified as follows:
   (a) The number of nodes on the element, and
   (b) The node numbers of the element's nodes as they appear in counterclockwise order.

The input is free format: Columns and line (card or record) boundaries are not significant, and numbers are separated by an arbitrary number of blanks (or line boundaries). For example, the input in Fig. 8.2 defines the mesh in Fig. 8.3, consisting of eight nodes, four triangular elements, and one quadrilateral element.

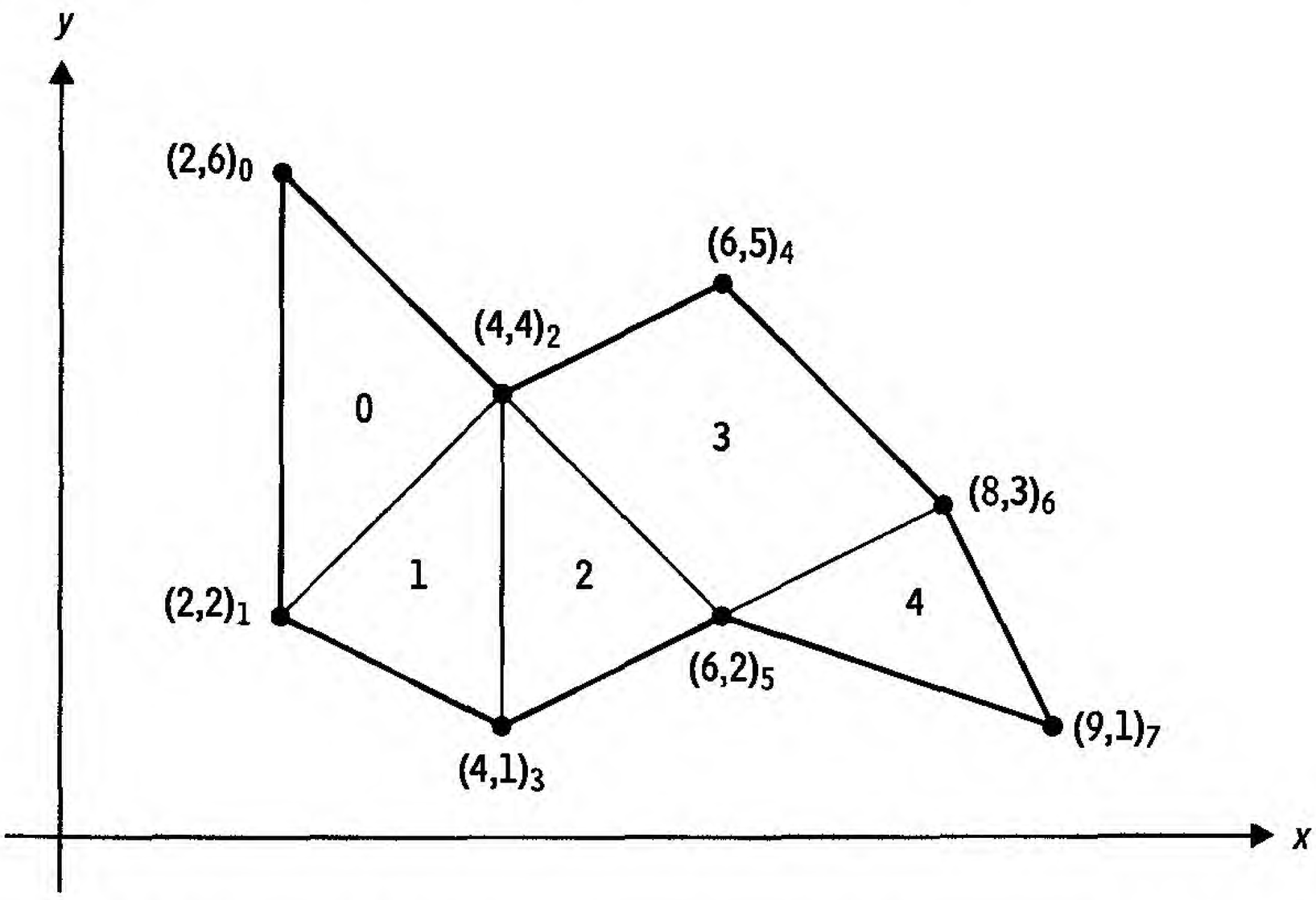


Figure 8.3 Example Mesh. The boundary of the domain is shown in boldface lines, and nodes are shown as solid dots. Node numbers are indicated as subscripts to node coordinates. Element numbers appear at the center of each element.

The major requirements for the representation are as follows:

1. Practical problems can have tens or even hundreds of thousands of nodes and elements. Therefore the representation must be compact.
2. Nodes and elements are used, together with other information, to construct matrix equations, which are then solved for the values of the dependent variables at the nodes. To construct the matrices, it must be possible to find quickly the coordinates of a node and the nodes of an element.

The finite element solution process itself is too complex to use as an example here. Although it is difficult to characterize mesh quality, it is generally agreed that elements should not have large angles. Therefore we will exercise our code by checking for elements having an angle that exceeds a threshold.

We will develop and use three programs, pausing after developing each for a critique, and evolving the code in response. As you work through our code, we suggest that you keep the following questions in mind:

1. Is the code readable and understandable? Is the overall structure apparent without understanding the details?
2. Suppose the representation must be changed to admit some new use. How much existing code that uses the representation would need to be changed?

As you think about these questions, bear in mind that a real finite element program would have vastly more code that uses the representation: Extrapolate your conclusions!

8.2 Solution One: Arrays

Our first solution to the finite element input problem is a C++ analog to what one might write in FORTRAN or C.

8.2.1 Representation

Finite element programs written in FORTRAN often represent the mesh with two tables, the *node table* and the *element table*. Both are typically two-dimensional arrays. The coordinates of each node are stored in a column of the node table array. (Since FORTRAN arrays are stored in column-major order, storing the coordinates in a column keeps the coordinates of a node adjacent in memory). Each element is likewise stored in a column of the element table array as a count of the

0:	1:	2:	3:	4:	5:	6:	7:
2.0	2.0	4.0	4.0	6.0	6.0	8.0	9.0
6.0	2.0	4.0	1.0	5.0	2.0	3.0	1.0

Node Table

0:	1:	2:	3:	4:
3	3	3	4	3
1	3	5	6	7
2	2	2	4	6
0	1	3	2	5
			5	

Element Table

Figure 8.4 Example Node and Element Table. Each column of the node table contains the (x, y) coordinates of a node. Each column of the element table represents an element: The first row is the number of nodes in the element, and the remaining rows contain node numbers. A blank entry indicates a table element that has not been set. The element table shown can hold elements with a maximum of five nodes. Typical node and element tables for the example of Fig. 8.3 are shown.

number of nodes in the element, followed by the indices into the node table of the element's nodes in counterclockwise order. A node and element table that corresponds to the example of Fig. 8.3 is shown in Fig. 8.4. The input format, which was chosen to be consistent with the input formats used by existing finite element programs, mimics the internal node and element table representation.

8.2.2 Implementation

Since we are working in C++, not FORTRAN, we modify the representation slightly to take advantage of the `Point` class from Section 6.1 (page 144) and the `SimpleArray<T>` class template of Section 4.3, as modified in Exercise 6.2.

Our node table is a one-dimensional array of `Point` objects, a `SimpleArray<Point>`. Each element of the mesh can be represented as a one-dimensional array of `int` objects:

ch8/solution1.C

```
typedef SimpleArray<int> Element;
```

Our element table is thus a one-dimensional array of one-dimensional arrays of `int` objects, a `SimpleArray<Element>`. Since the number of items stored in a `SimpleArray<T>` object can be set and queried at runtime, we don't need to store explicitly the node count for each element. With these decisions made, code to read the input, build the tables, and call a function to check element angles is straightforward:

ch8/solution1.

```
int main() {

    // Read sizes and create tables
    int numNodes;
    int numElements;
    cin >> numNodes >> numElements;

    SimpleArray<Point> nodeTable(numNodes);
    SimpleArray<Element> elementTable(numElements);

// Read node data
for (int nodeNum = 0; nodeNum < numNodes; nodeNum++) {
    cin >> nodeTable[nodeNum];
}

// Read element data
for (int eltNum = 0; eltNum < numElements; eltNum++) {
    Element& e = elementTable[eltNum];   // e is an alias for element # eltNum
    int elementSize;
    cin >> elementSize;
    e.setSize(elementSize);
    for (int i = 0; i < elementSize; i++) cin >> e[i];
}

// Read maximum angle threshold and check elements
Number angle_threshold;
cin >> angle_threshold;
Boolean anglesOK = Boolean::true;
for (eltNum = 0; eltNum < numElements; eltNum++) {
    Element& e = elementTable[eltNum];  // e is an alias for element # eltNum
    if (maxAngle(e, nodeTable) > angle_threshold) {
        cout << "Element [ ";
        for (int i = 0; i < e.numElts(); i++) cout << nodeTable[e[i]] << " ";
        cout << "] has a large angle." << endl;
        anglesOK = Boolean::false;
    }
}
    return anglesOK ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

The function for checking element angles uses the angle() member function added to Point in Exercise 4.3:

ch8/solution1.C

```
Number
maxAngle(const Element& e, const SimpleArray<Point>& nodeTable) {
    Number maxang = 0.0;
    int numNodes = e.numElts();
    for (int i = 0; i < numNodes; i++) {
        int ccwNodeNum = (i + 1) % numNodes;
        int cwNodeNum = (i + numNodes - 1) % numNodes;
        Number angle = nodeTable[e[i]].angle(
                                        nodeTable[e[cwNodeNum]],
                                        nodeTable[e[ccwNodeNum]]
                                    );
        if ( angle > maxang ) maxang = angle;
    }
    return maxang;
}
```

For each node in the element, angle() is called with the node's clockwise and counterclockwise neighbors as arguments. The node numbers are computed using the modulus operator (%), which computes the remainder of dividing its left operand by its right. If either operand of the modulus operator is negative, the result is implementation dependent, so we compute the node number of the clockwise neighbor by adding numNodes-1 instead of subtracting 1. ARM §5.6

8.2.3 Critique

The code is readable and understandable. The overall structure is readily apparent by reading the comments and skipping the code. The program does exactly the job it is supposed to do. If that is all that matters, we would be happy and done. Indeed in some situations success means getting a specific task done quickly and simply; once the job is done, the program is thrown away. But more often, a program must grow and evolve to meet additional or changing requirements.

With our small program, evolution doesn't cost much. At worst, you throw away the program and start over: It is only about 50 lines of code. But a real finite element program would be much, much larger, with commercial packages consisting of many tens of thousands of lines of code. To critique our program meaningfully, we ignore the possibility of starting over, which exists with small programs but not with larger programs. (Of course, there is no sharp boundary between small and large. Rather there are trade-offs between the cost of change and the benefits of the change. With a program as small as ours, the cost of change

is so low that efforts to make change less expensive are not worthwhile. You must extrapolate to the programs you work with in practice.)

Our program performs two tasks: building the node and element table representation of the mesh from the input data, and checking each element against a maximum angle threshold. The first three "paragraphs" of `main()` build the representation, while the last "paragraph" and the `maxAngle()` function use the representation. In a real program the balance would shift: A small amount of code would build the representation and more code would use it.

Suppose providing new features required changing the representation. We would expect to have to change or even rewrite the code that builds the representation. In our program, we might also have to change all the code that uses the representation. The angle-checking code depends on the following assumptions:

1. There is only one mesh.
2. There is an element table that is indexed by element number.
3. There is a node table that is indexed by node number.
4. Nodes of an n-node element are indexed with respect to individual elements such that the neighbors of the node indexed i are indexed $i + 1 \bmod n$ (counterclockwise) and $i - 1 \bmod n$ (clockwise).

Invalidate any of these assumptions and the angle-checking code has to be changed. Extrapolating to larger programs, we conclude that our program doesn't meet the challenge of change.

We shall develop a solution more amenable to change in Section 8.4. First we pause to discuss some general principles.

8.3 Abstraction and Encapsulation

Our first solution relies on functions as subroutines. A good subroutine is intended to be an abstraction, to perform some useful function that can be described concisely, without recourse to the subroutine's code. *Abstraction*, considering the important and ignoring the less important, is fundamental to analytic thought. In the words of the naturalist D'Arcy Thompson [112, page 271],

> we must learn from the mathematician to eliminate and to discard; to keep the type in mind and leave the single case, with all its accidents, alone; and to find in this sacrifice of what matters little and conservation of what matters much one of the peculiar excellences of the method of mathematics.

Mathematical library routines illustrate abstraction at work: We can describe concisely the function of `sqrt()` without knowing the irrelevant detail of how it

works. We write subroutines precisely because they hide irrelevant detail. Substitution of the concise for the complex helps us cope with complexity. When we violate a subroutine's abstraction by using knowledge of its implementation, we reduce our ability to manage complexity.

The subroutine abstraction works for simple transformations: We find little need to examine the details of sqrt(). However, when data must be shared among many routines via long argument lists and COMMON blocks (FORTRAN) or global variables (C and C++), then we often need to examine many subroutines in detail to understand the system. The subroutine abstraction fails for manipulations of complicated data.

Classes combine local data and groups of subroutines operating on that data to give a new form of abstraction. Much more complex systems of data and algorithms can be handled by classes in a way that is more localized than comparable formulations based on functions alone.

Finding good abstractions is an intellectual challenge. An important guide is Parnas's *information hiding* principle [87, 88, 90]:

- Hide from an abstraction's clients design decisions that are likely to change.

The hidden decisions are often called the abstraction's *secrets*. Perhaps the most important application of the information hiding principle is the following adage:

- Keep data structures secret.

Data structures that are used to implement an abstraction are both irrelevant to its clients and likely to change. Brooks observes [17, page 102]:

> Beyond craftsmanship lies invention, and it is here that lean, spare, fast programs are born. Almost always these are the result of strategic breakthrough rather than tactical cleverness. Sometimes the strategic breakthrough will be a new algorithm, . . . [but] [m]uch more often, strategic breakthrough will come from redoing the representation of the data or tables. This is where the heart of a program lies.

To produce such "lean, spare, fast programs," we must keep the cost of changing data structures low. This can only be achieved by hiding an object's data structures from its clients: If a client may depend on an object's data structures, changing the data structure will cost at least the time to understand the client's code, and maybe more.

Preventing access to an object's implementation, including its data structures, is called *encapsulation*. The basic C++ mechanism for encapsulation is the distinction between public and private class members: public class members can be accessed

by any function; private members can only be accessed by members of the same class and by friends (see Section 6.10). Private members are the class's secrets. Public members are the interface between the class and its clients. In effect, the public members constitute a contract [116] that defines the permissible interactions among the class and its clients.

The only abstraction expressed in our program of Section 8.2 is a point in two-dimensional space, as represented by the Point class. We have given the name Element to an array of points, but this is not an abstraction, for the details are exposed. The representations we have chosen are visible everywhere. There is no information hiding. It is this lack of abstraction and information hiding that makes (an extrapolation of) this program expensive to change. We now take our first steps to remedy this problem.

8.4 Solution Two: Introducing Classes

In this section, we rework our program of Section 8.2 to express in code the abstractions we only discussed in prose there. The basic representation remains the same: A mesh is represented by a node table and an element table, with each element represented by an array of node numbers, and each node represented by a Point object.

8.4.1 Classes

We know from the problem description and our previous solution that we will want to have node and element objects, and that these will somehow be organized into meshes. We use the plural meshes because there is no reason to presume that programs manipulate only one mesh at a time.

For the moment, a node is nothing more than a Point. However, a node is a key idea in a finite element manipulation program. Not all Point objects are meant to be node objects.

- Give names to abstractions.

To clarify our code, we give the name Node to Point:

ch8/solution2.C

```
typedef Point Node;
```

This makes it easy to add function to Node: Write a Node class that behaves like a Point but has the additional function; then recompile the program. For example, one can imagine wanting to find all of the nodes that are neighbors in the mesh of a given node; see Exercise 8.9.

As before, we represent an element as an array of nodes, but this time we encapsulate the representation in a class:

ch8/solution2.C

```
class Element {
public:
    int operator[ ](int i) const;
    int numNodes()      const;
    friend istream& operator>>(istream&, Element&);
private:
    SimpleArray<int> node_numbers;
};
```

In Section 8.2 we used a typedef to give the name Element to an array of integers; here we create an Element class. Doing so allows us to define an input operator for elements and a reasonable way to query the number of nodes in an element (instead of the incongruous expression e.numElts() used in the maxAngle() function on page 205). The implementations are simple:

ch8/solution2.C

```
int Element::operator[ ](int i) const { return node_numbers[i];         }
int Element::numNodes()      const { return node_numbers.numElts(); }

istream& operator>>(istream& instream, Element& e) {
    int n;
    instream >> n;
    e.node_numbers.setSize(n);
    for (int i = 0; i < n; i++) instream >> e.node_numbers[i];
    return instream;
}
```

Notice that Element does not provide a maxAngle() member function. Element is the natural place for it, but since an element only stores node numbers, an Element object can't get the relevant Node objects without knowing where the node table is. That information could be passed to maxAngle(), but we chose instead to put maxAngle() in the Mesh class instead.

A Mesh object represents a mesh with node and element tables:

ch8/solution2.C

```
class Mesh {
public:
    Boolean checkElementAngles(Number angle_threshold) const;
    Number maxAngle(const Element& e) const;
    int numNodes() const;
    int numElements() const;
    friend istream& operator>>(istream&, Mesh&);
private:
    SimpleArray<Node>    node_table;
    SimpleArray<Element> element_table;
};
```

Putting these tables in a class allows a program to manipulate several meshes simultaneously. Further, the Mesh input operator handles the details of reading in a mesh. The maxAngle() member is the maxAngle() function from Section 8.2, altered to be a member of Mesh by accessing the node table in the Mesh object for which it is called:

ch8/solution2.C

```
Number Mesh::maxAngle(const Element& e) const {
    Number maxang = 0.0;
    int numNodes = e.numNodes();
    for (int i = 0; i < numNodes; i++) {
        int ccwNodeNum = (i + 1) % numNodes;
        int cwNodeNum = (i + numNodes - 1) % numNodes;
        Number angle = nodeTable[e[i]].angle(
                                              nodeTable[e[cwNodeNum]],
                                              nodeTable[e[ccwNodeNum]]
                                             );
        if ( angle > maxang ) maxang = angle;
    }
    return maxang;
}
```

Likewise, the checkElementAngles() member is the code from the bottom of the main() function in Section 8.2 that has been modified to access the element table from the Mesh object:

ch8/solution2.C

```
Boolean Mesh::checkElementAngles(Number angle_threshold) const {
    Boolean anglesOK = Boolean::true;
    for (int eltNum = 0; eltNum < numElements(); eltNum++) {
        const Element& e = element_table[eltNum];   // e is an alias for element # eltNum
        if (maxAngle(e) > angle_threshold) {
            cout << "Element [ ";
            for (int i = 0; i < e.numNodes(); i++) cout << node_table[e[i]] << " ";
            cout << "] has a large angle." << endl;
            anglesOK = Boolean::false;
        }
    }
    return anglesOK;
}
```

Since we have moved the node and element tables from the caller into Mesh, we can provide an input operator:

ch8/solution2.C

```
istream& operator>>(istream& instream, Mesh& m) {
    int nNodes;
    int nElements;
    instream >> nNodes >> nElements;
    m.node_table.setSize(nNodes);
    m.element_table.setSize(nElements);
    for (int nodeNum = 0; nodeNum < nNodes; nodeNum++) {
        instream >> m.node_table[nodeNum];
    }
    for (int elementNum = 0; elementNum < nElements; elementNum++) {
        instream >> m.element_table[elementNum];
    }
    return instream;
}
```

Now essentially all of the work is done in the class member functions; the main() function becomes trivial:

ch8/solution2.C

```
int main() {
    Mesh m;
    cin >> m;

    Number angle_threshold;
    cin >> angle_threshold;
    Boolean anglesOK = m.checkElementAngles(angle_threshold);

    return anglesOK ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

8.4.2 Critique

This solution puts abstractions—node, element, and mesh—into our code. The most obvious benefit is the clarity and conciseness of the main() function: All the details are elsewhere. Contrast this with our first solution, in which the overview was expressed in comments, with the details intermixed. (Again, extrapolate to larger programs.) Since the tables are now in Mesh, not in the main() function, a program can work with multiple meshes simultaneously.

This solution also uses encapsulation: The data used to represent elements and meshes are held as private members of the respective classes. Encapsulating data as private members of a class can simplify debugging because you know that the data are only used by members of the class or its friends (except, of course, for damage done by out-of-range subscripts and pointers).

However, all is not well. The symptoms are apparent in the public interface to Element. The function for computing the maximum angle in an element is missing; it is a member of Mesh, not of Element as one would expect. The subscript operator ([ ]) does not fit an abstraction of an element: It takes an integer argument and returns an integer value. How are these relevant to elements? They're not. They are only relevant to the implementation of meshes with node and element tables. Abstractly, an element consists of an ordered sequence of nodes, which may or may not be stored as an array.

Providing access to an array data member via a member function that takes an index as argument exposes the array implementation. Before providing such a member function, ask whether other reasonable representations would admit an efficient implementation of the member function.

- Unless arraylike behavior is part of the abstraction that a class implements, avoid providing a member function that takes or returns an integer index argument.

Although we have employed encapsulation in this solution, we have not done a good job of information hiding: Most of the code exploits knowledge of the array-based representation. (See Exercise 8.7.) We now tackle information hiding.

8.5 Solution Three: Information Hiding

The root of the problems with our previous solution is that we didn't get the abstractions right. We developed abstractions of the things—meshes, nodes, and elements—but not of the relationships among them: Nodes are in elements, elements are in meshes, and, for communicating with the outside world, there is a mapping between nodes and positive integers and between elements and positive integers. We expressed these relationships by exposing everywhere the array-based representation, thus violating the information hiding adage to keep data structures secret.

8.5.1 Classes

We drive the design of our revised solution by asking what are the essential aspects of each abstraction. Let's start with Mesh. The problem statement tells us that a mesh contains nodes and elements, and that a mesh can be read from an input stream. Clients need to be able to get at all of the elements and perhaps all of the nodes in a mesh.

Consider the checkElementAngles() member function of the Mesh class in Section 8.4. It can be viewed as a client of the mesh abstraction even though it is a member function: It is conceptually independent of the mesh representation, and

if we change the mesh representation we shouldn't have to change checkElement-Angles(). As implemented on page 210, checkElementAngles() exploits directly knowledge of the mesh representation; it therefore must be changed if the representation is changed.

To avoid such unnecessary coupling in our revised solution, we must provide a more abstract way to access the mesh's elements and nodes: We'll use iterators (cf. Section 6.10) called NodesOfMesh and ElementsOfMesh. Here is the resulting Mesh class:

ch8/solution3.C

```
class Mesh {
public:
    friend ElementsOfMesh;      // Iterator over elements of a mesh
    friend NodesOfMesh;         // Iterator over nodes of a mesh
    friend istream& operator>>(istream&, Mesh&);
    friend NodeReader;

    Boolean checkElementAngles(Number angle_threshold) const;
private:
    SimpleArray<Node> node_table;
    SimpleArray<Element> element_table;
};
```

The NodeReader class is described after we discuss the Element class.

As before, we will represent a node as a point:

ch8/solution3.C

```
typedef Point Node;
```

(However, see Exercise 8.9.) An element consists of an ordered sequence of nodes, without reference to node or element numbers. Again, we need to hide our choice of representation for an element by providing an abstract way to access the nodes of an element; Element uses an iterator to provide such access:

ch8/solution3.C

```
class Element {
public:
    friend NodesOfElement;    // Iterator over nodes of an element

    friend void operator>>(NodeReader& reader, Element& e);
    friend ostream& operator<<(ostream& os, const Element& e);

    Number maxAngle() const;
private:
    SimpleArray<Node*> node_ptrs;
};
```

Since an Element should not know the mapping between node numbers and Node objects, an Element is represented by an array of pointers to Node objects rather than an array of node numbers. This insulates Element from changes in the way Mesh stores nodes. However, it also makes it impossible for an Element object to read itself from an input stream because it doesn't know about the node numbers that appear in the input format. A partnership between Mesh and Element is needed to resolve this problem; NodeReader objects mediate between Mesh and Element:

ch8/solution3.C

```
class NodeReader {
public:
    NodeReader(Mesh& m, istream& instream);
    int getSize();
    Node* getNode();
private:
    Mesh& mesh;
    istream& in;
};
```

A NodeReader is created by the Mesh reading function:

ch8/solution3.C

```
istream& operator>>(istream& instream, Mesh& m) {
    // Set sizes of node and element tables
    int nNodes;
    int nElements;
    instream >> nNodes >> nElements;
    m.node_table.setSize(nNodes);
    m.element_table.setSize(nElements);

    // Read nodes
    for (int nodeNum = 0; nodeNum < nNodes; nodeNum++) {
        instream >> m.node_table[nodeNum];
    }

    // Read elements
    NodeReader reader(m, instream);
    for (int elementNum = 0; elementNum < nElements; elementNum++) {
        reader >> m.element_table[elementNum];
    }
    return instream;
}
```

The NodeReader object is initialized with a reference to the Mesh being read and the stream it is being read from:

ch8/solution3.C

```
NodeReader::NodeReader(Mesh& m, istream& instream) :
    mesh(m), in(instream) {}
```

As a friend of Mesh, a NodeReader knows both the mesh input format and how to map a node number read from the input stream to a pointer to a Node:

ch8/solution3.C

```
int NodeReader::getSize() {
    int size;
    in >> size;
    return size;
}

Node* NodeReader::getNode() {
    int nodeNum;
    in >> nodeNum;
    return &mesh.node_table[nodeNum];
}
```

These capabilities can then be used to implement reading an element:

ch8/solution3.C

```
void operator>>(NodeReader& reader, Element& e) {
    int nNodesInElement = reader.getSize();
    e.node_ptrs.setSize(nNodesInElement);
    for (int i = 0; i < nNodesInElement; i++) {
        e.node_ptrs[i] = reader.getNode();
    }
}
```

From the standpoint of information hiding, Mesh and NodeReader share two secrets: knowledge of the input format and of the way that Node objects are stored. NodeReader provides an abstraction of those secrets to Element. The relationship is not symmetric: NodeReader does not have access to the secrets of Element. Indeed NodeReader is a friend of Mesh but not of Element.

Now that we've seen how an Element gets read from a stream, let's look at how a client accesses the elements of a mesh. An ElementsOfMesh object is an iterator over the elements of a mesh:

ch8/solution3.C

```
class ElementsOfMesh {
public:
    ElementsOfMesh(const Mesh& m) :
        element_table(m.element_table), cur(m.element_table.numElts()-1) { }

    Boolean          more()    const { return cur >= 0;          }
    void             advance()       { --cur;                   }
    const Element& current() const { return element_table[cur]; }
```

```
private:
    const SimpleArray<Element>& element_table;
    int cur;
};
```

The connection between a mesh and its ElementsOfMesh iterator is established in the Elements constructor, called when the iterator is initialized.

As a friend of Mesh, ElementsOfMesh has knowledge of how Mesh stores elements; specifically, an ElementsOfMesh object holds a reference to the element_table of the Mesh with which it was initialized. The cur member datum holds the state of the iteration; it is the analog of a loop control variable. The current() member function returns a reference to the current element of the iteration, more() tests whether additional elements remain to be iterated, and advance() steps to the next element to be iterated.

Here is an example of how ElementsOfMesh can be used:

ch8/solution3.C

```
Boolean Mesh::checkElementAngles(Number angle_threshold) const {
    Boolean anglesOK = Boolean::true;
    for (ElementsOfMesh elts(*this); elts.more(); elts.advance()) {
        if (elts.current().maxAngle() > angle_threshold) {
            cout << "Element " << elts.current() << " has a large angle." << endl;
            anglesOK = Boolean::false;
        }
    }
    return anglesOK;
}
```

The variable elts is an ElementsOfMesh object initialized with *this, meaning the Mesh object used to select and call the checkElementAngles() member function. Thus for Mesh m, m.checkElementAngles(.5) creates an ElementsOfMesh iterator over the elements of Mesh m, checking each element for angles over 0.5 radians.

In a similar manner, NodesOfElement iterates over the nodes of an element. There is one crucial difference: There is an ordering among the nodes of an element, but not among the elements of a mesh. One could provide multiple iterators, perhaps one that iterates the nodes in clockwise order, one counterclockwise, and one in arbitrary order. For simplicity, we've provided only one iterator, which provides access to both neighbors of the current iterate:

ch8/solution3.C

```
class NodesOfElement {
public:
    NodesOfElement(const Element& e) :
        node_ptrs(e.node_ptrs), cur(e.node_ptrs.numElts()-1) {}
```

```
// Iteration
    Boolean more()    const { return cur >= 0;          }
    void    advance()       { --cur;                    }
    Node&   current() const { return *node_ptrs[cur]; }

// Neighbors of current iterate
    Node& ccwNeighbor() const { return *node_ptrs[(cur + 1) % node_ptrs.numElts()]; }
    Node& cwNeighbor() const {
        int numNodes = node_ptrs.numElts();
        return *node_ptrs[(cur + numNodes - 1) % numNodes];
    }
private:
    const SimpleArray<Node*>& node_ptrs;
    int cur;
};
```

The constructor and iteration control members are similar to the corresponding members of ElementsOfMesh. The neighbor members know how an Element stores pointers to its nodes and use this knowledge to compute neighbors. Their use is illustrated by the function that computes the maximum angle in an element:

ch8/solution3.C

```
Number Element::maxAngle() const {
    Number maxang = 0.0;
    for (NodesOfElement nodes(*this); nodes.more(); nodes.advance()) {
        Number angle = nodes.current().angle(nodes.cwNeighbor(), nodes.ccwNeighbor());
        if ( angle > maxang ) maxang = angle;
    }
    return maxang;
}
```

The classes we have developed in this section can be exercised using the main() function from page 211.

8.5.2 Critique

We developed this solution by applying the information hiding principle to the design of the abstractions used in the program. In particular, we introduced classes that represent relationships—nodes in meshes, elements in meshes, and nodes in elements—and used those classes to hide our choice of data structures. In part this solution demonstrates that there need not be a one-to-one correspondence between abstractions and classes. For example, the NodesOfMesh and ElementsOfMesh classes are very much a part of the mesh abstraction. Indeed the close

coupling between these classes and the Mesh class is indicated by the friend declarations in Mesh.

Measured by length, or number of statements (including declarations), or number of classes, this solution is more complex than the previous two solutions. Designing abstractions is difficult, and it is uncommon to get them right the first time. The extra work is repaid as the program evolves; if the program is written and used once and then thrown away, the extra work is wasted. The vast majority of programs change to meet changing requirements or because the act of developing the program leads us to a better understanding of the requirements. For programs that change, the work is repaid in lower cost of change. The payoff scales (nonlinearly) with the size of the program.

A number of the exercises are designed to demonstrate (on a small scale) the benefits of applying information hiding. We urge you to try them.

8.6 Notes and Comments

8.1 Two classic texts on the finite element method are by Zienkiewicz [124] and by Bathe [11]; a particularly concise description of the method can be found in [55, Chapter 8]. Input modules for finite element programs of realistic complexity are described in [11, Appendix], [34, Chapter 6], and [124, Chapter 24].

8.2 This chapter does not illustrate the advantages of C++ over FORTRAN-90; in fact, the style of C++ programming in this chapter is comparable to what one could write in FORTRAN-90 [79, 78].

8.3 Parnas, in now classic papers [87, 88], first proposed information hiding as a criterion for partitioning programs into modules. His ideas have had a profound impact on the practice of software engineering and the development of object-oriented program design.

8.4 Booch [16] stresses the importance of abstraction and encapsulation in object-oriented programming and describes techniques for finding good abstractions. Chapter 3 of his book features an extensive discussion of the various roles and interrelationships among classes, objects, and member functions. Based on these roles and interrelationships, he offers what he calls "heuristics" for choosing operations (i.e., functions), relationships among classes and objects, and implementations (i.e., class and function definitions). A more detailed, but narrower, view of the interplay between classes and of the role of functions in classes appears in [54].

8.5 Some authors distinguish between encapsulation and information hiding; others don't. Booch [16, page 513] defines them both to be "the process of hiding all of the details of an object that do not contribute to its essential characteristics." We call this process information hiding. We use the term encapsulation to mean a language mechanism that prevents access to certain information. See also [92, Section 5-2].

8.6 Encapsulation in C++ helps a programmer produce robust code. It does not function as a security mechanism: private members are both visible in the class definition and accessible via subterfuge. Any simple mechanism preventing human readers from accessing private members prevents compilers from accessing this information as well, leading to unacceptable runtime overhead in some cases. Accessors (Section 13.6.1) or other classes delegating their member functions to a pointer can hide implementation completely from the readers of source code if this is essential. However, reading implementations unfortunately remains one of the more effective ways of learning the meaning of classes for many software systems: Efforts to hide implementation details completely must be coupled with interface documentation.

8.7 Exercises

8.1 Explain where in the code of Section 8.2 each of the assumptions listed on page 206 is made.

8.2 Implement the NodesOfElement class from Section 8.5.

8.3 The program of Section 8.5 uses an input operator (operator>>()) that reads directly from a stream into a Point, but the input operator for Element does not read directly from a stream. Explain the essential difference between Point and Element that motivated this decision.

8.4 Identify the secrets of each class in the program of Section 8.5.

8.5 In some applications of the finite element method, a so-called *structured mesh* (or *structured grid*) can be used. In a structured mesh, the nodes are stored in a rectangular array and the elements are defined implicitly by the array of nodes. Given a pair of subscripts, let $n_{i,j}$ denote the node at the (i, j)th position of the array. The (i, j)th element then consists of the nodes $n_{i,j}, n_{i+1,j}, n_{i+1,j+1}, n_{i,j+1}$. Modify the program of Section 8.4 to exploit this structure. Which functions remained unaltered?

8.6 Repeat Exercise 8.5 for the program of Section 8.5.

8.7 It is sometimes necessary to refine a mesh by adding nodes and elements. A simple refinement technique (which, unfortunately, can lead to large angles) is to replace each element with several elements formed by adding a node positioned at the element's centroid and forming triangular elements that connect the new node to each edge of the original element. Applying this refinement technique to the mesh of Fig. 8.3 yields the mesh of Fig. 8.5. Letting (x_i, y_i) denote the coordinates of node i, $0 \leq i < n$, the centroid has coordinates

$$\left(\frac{1}{n} \sum_{i=0}^{n-1} x_i, \frac{1}{n} \sum_{i=0}^{n-1} y_i \right).$$

It is difficult to implement refinement efficiently with an array-based representation. Why? Change the program of Section 8.4 to use lists (cf. List<T> on page 170 and your

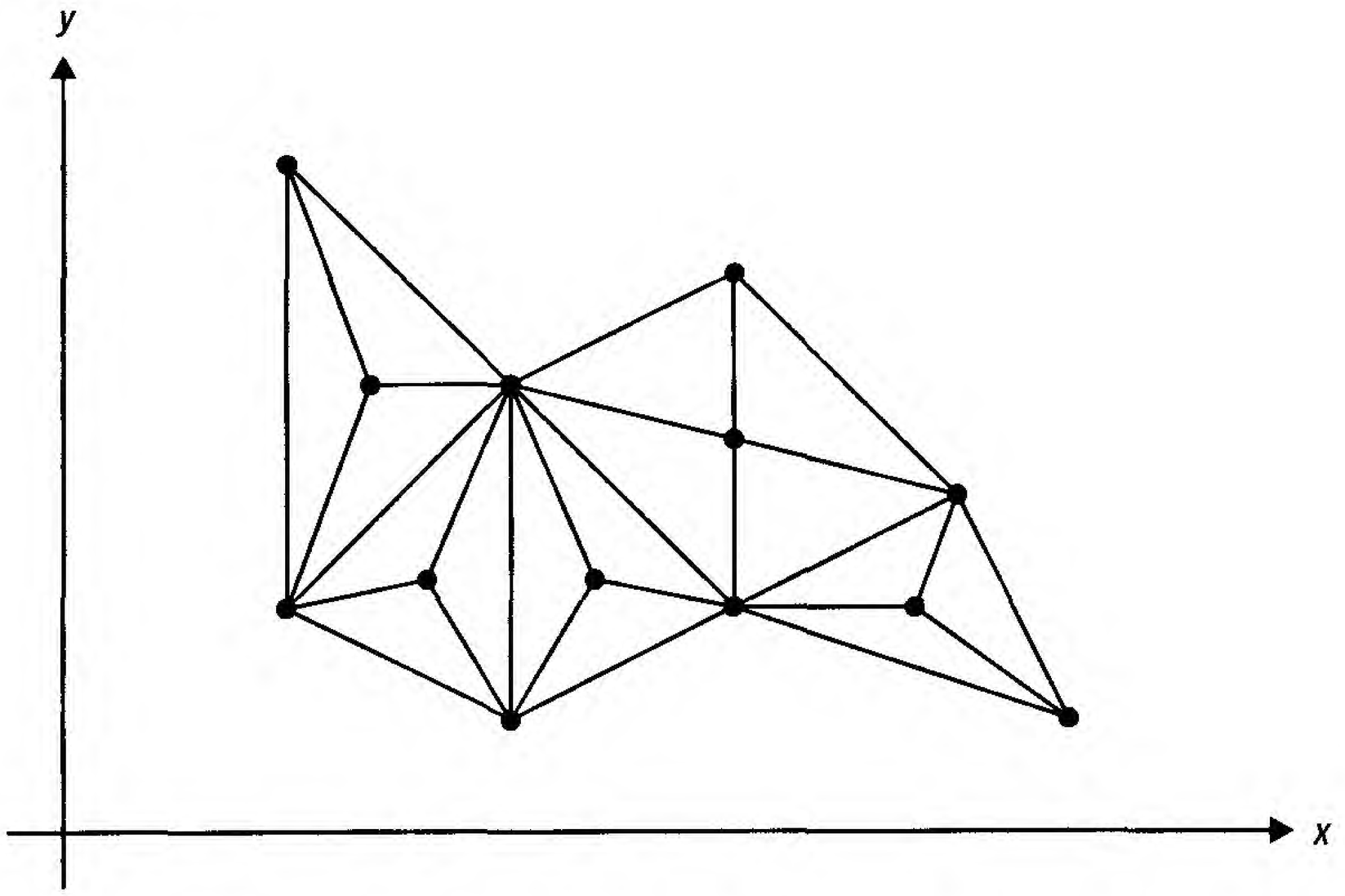


Figure 8.5 Result of Applying the Refinement Technique Described in Exercise 8.7 to the Mesh in Fig. 8.3.

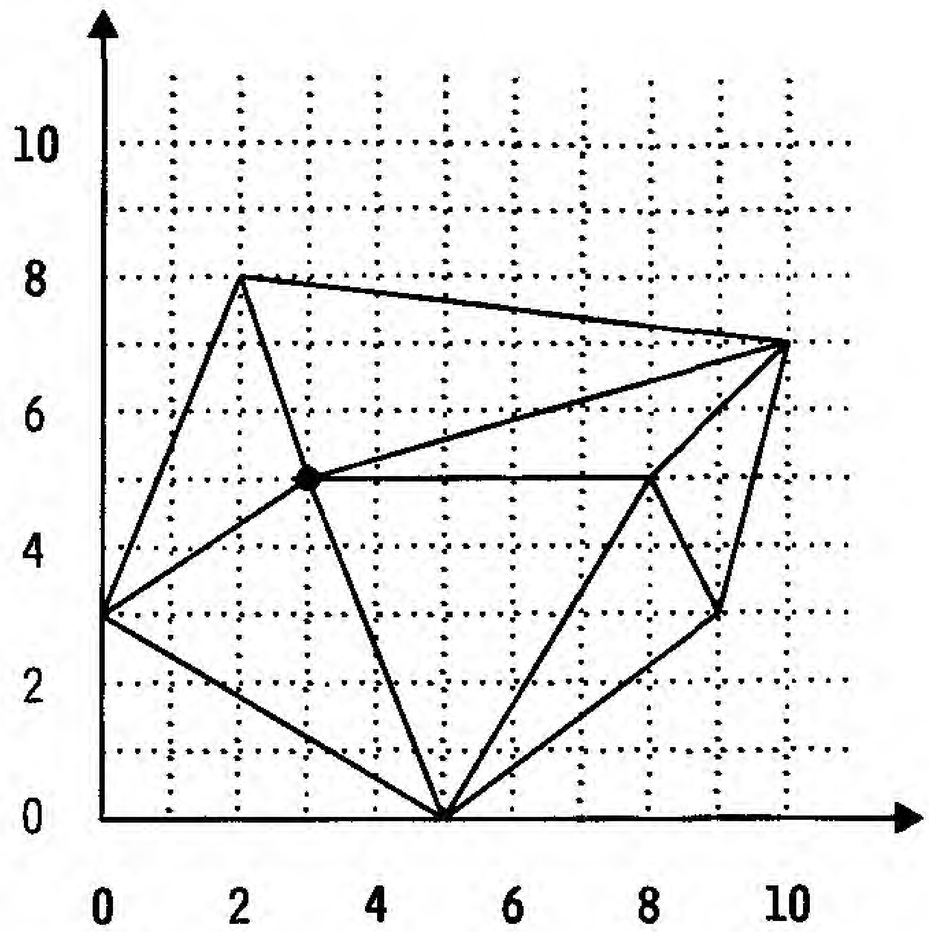
solution to Exercise 6.14) instead of arrays. Add a refine() member function to your modified Mesh class. Which functions remained unaltered?

8.8 Repeat exercise Exercise 8.7 for the program of Section 8.5.

8.9 It is common practice to apply to a mesh a process called *Laplacian smoothing*. In its simplest form, a step of Laplacian smoothing selects an interior node and moves that node to the centroid (see Exercise 8.7) of the node's neighbors (two nodes are neighbors if they are connected by an edge). One step of Laplacian smoothing is illustrated in Fig. 8.6. The process is repeated once for each interior node, in arbitrary order.

Modify the program of Section 8.5 to implement Laplacian smoothing as follows. Write a Node class that, in addition to behaving like a Point, provides access to its neighbors. Then add a member function to Mesh that implements Laplacian smoothing using the Node class; this member should be immune to changes in the mesh representation.

8.10 Consider your solution to Exercise 8.9. Meshes encountered in practice may have tens of thousands of nodes, and Laplacian smoothing may be applied to the mesh several times. Does your solution recompute the node neighbor information each time, or does it compute it once and save it? The first approach costs time, the second approach costs space. Suppose that after measuring the performance of your program, you decided that you made the wrong choice. Discuss the impact on your program of changing to the other approach.



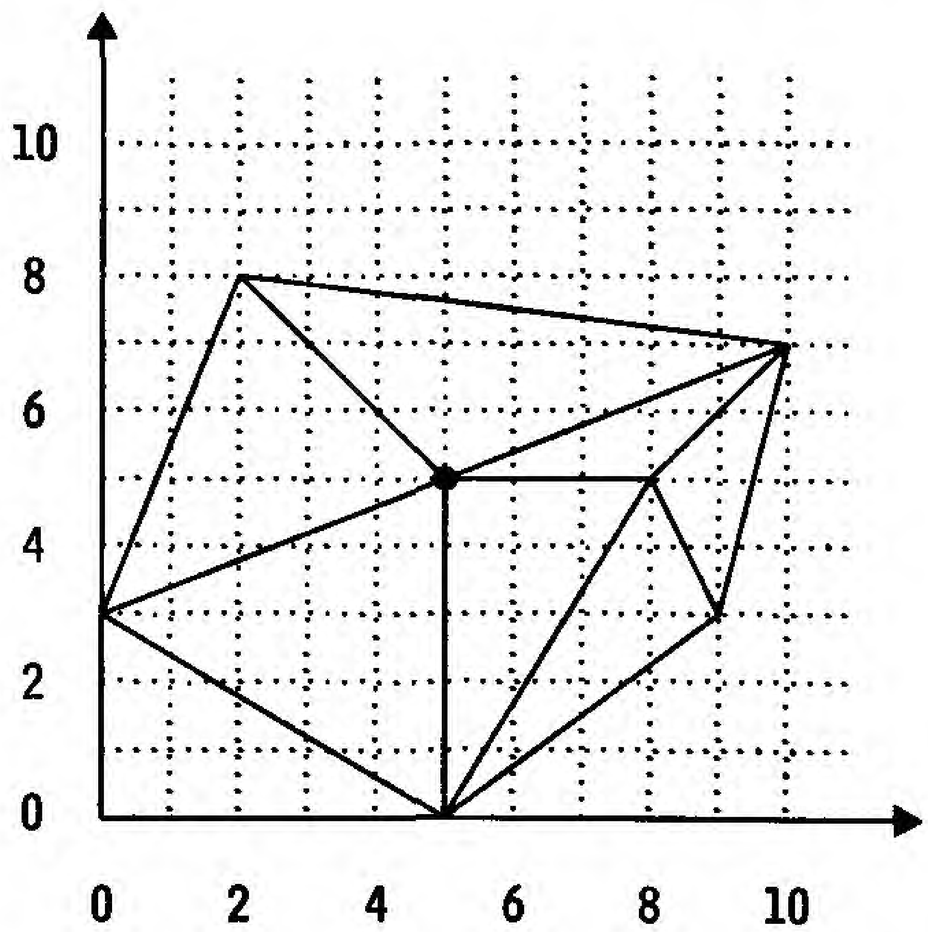


Figure **8.6** Laplacian Smoothing. The original mesh is shown on the left. Laplacian smoothing applied to the node marked with a solid dot yields the mesh on the right.

PART II

Expressing Commonality

CHAPTER 9

Expressing Common Behavior

In this chapter, we introduce virtual member functions and the concept of an interface category, a group of classes with some common virtual member function declarations. These virtual function declarations act like standard connections, allowing objects from classes in this category to be operated through the interface. Virtual functions are central to object orientation in C++.

We begin in Section 9.1 with a brief description of the running example used in this chapter and the next. Section 9.2 reviews classes and objects, concentrating on how encapsulation and information hiding make programs easier to change. Section 9.3 introduces the idea of interface, motivated by an example of two classes sharing "is-usable-as" commonality, and virtual functions, the C++ mechanism for capturing "is-usable-as" commonality. Section 9.4 introduces a way of sketching objects and of graphing the relations between classes in an interface category. Section 9.5 reviews the virtual function mechanism by contrasting the creation of objects as instances of a class with the use of objects through interfaces. Section 9.6 describes the relevant C++ mechanisms. Section 9.7 demonstrates classes with more than one interface, Section 9.8 demonstrates using interfaces as object components, and Section 9.10 demonstrates using interfaces to exceptions.

9.1 Example: Instrument Control

Throughout this chapter we adopt an animated view of objects, attributing to our C++ objects qualities of physical objects. We reinforce this view by using computer control of pretend electronic instruments as our running example. Then object control becomes instrument control. For example, we will discuss an `Acme130` object that allows us to write code to control an Acme (a fictitious brand) model 130 voltage supply from the computer. The `Acme130` object is not a voltage supply; it's just some computer bits and some functions using them that allow us to control the physical voltage supply from a computer. Nevertheless, this view of

objects as almost-real entities helps to form a new mental model of computing, to get "object oriented."

To be concrete, we pretend that the instruments in our example can be accessed through a *General Purpose Interface Bus*, or *GPIB* [26], the IEEE standard (IEEE-488) for computer-controlled measurement and test instruments. GPIB-attached instruments connect via a cable to an interface board in the computer and are controlled by passing control commands and data to functions supplied by the interface board manufacturer. The functions then transmit commands and data on the bus to the instruments. To route the commands to the proper instruments, each GPIB-attached instrument is given an integer address between 0 and 31. There are thousands of off-the-shelf products that have GPIB interfaces, and there are GPIB interface boards for most personal computers and workstations.

Ours is a toy example, ignoring many complexities of real GPIB systems. In fact, we will distill the entire GPIB connection down to a single GPIBController_Stub class with four member functions: insert() initializes a connection to a device, receive() obtains data from a device, and two versions of send(), send commands and send data to a device. Here is the class, with its members given inline for convenience:

ch9/GPIBController_Stub.h

```
class GPIBController_Stub {
public:
    void insert(const char* device_name, unsigned int address) {
        cout << "(" << device_name << " now at address " << address << ")" << endl;
    }
    void send(unsigned int address, const char* cmd) {
        cout << "(" << "GPIB instrument #" << address <<
                                                    " sends " << cmd << ")" << endl;
    }
    void send(unsigned int address, float f) {
        cout << "(" << "GPIB instrument #" << address <<
                                                  " sends value " << f << ")" << endl;
    }
    float receive(unsigned int address) {
        cout << "(" << "Please enter number for GPIB instrument #" << address << ": ";
        float f;
        cin >> f;
        cout << ")" << endl;
        return f;
    }
};
```

GPIBController_Stub represents the computer's GPIB controller board. The keyboard and display are its "bus."

The exercises lead you through the construction of a simple GPIB system simulator using a more interesting toy GPIB controller class. The classes developed in the simulator parallel closely the classes we discuss here.

Many of the other classes we develop in this chapter will appear in multiple versions: Evolution in the design and implementation of classes generates multiple versions of classes. To make it clear which version of each class we mean, we will rename the classes as we proceed, attaching suffixes separated by underscores. Thus our first version of a class to represent a GPIB controller is GPIBController_Stub, rather than just GPIBController. The _Stub on the end of this class name indicates that this class is meant to simulate GPIB controllers in a trivial way. In practice, classes are revised without necessarily changing their names and older versions are discarded or held by a source code control system.

9.2 Classes and Objects

Programs that are used evolve; therefore, C++ has features that—used effectively—aid revision. This chapter reviews one such tool, encapsulation, and introduces a new one, the virtual function interface. We start in this section with a review to introduce our example.

A programming object represents either a physical object, like a voltage supply, or a mental concept, like a matrix. Object-oriented programming creates and manipulates these representations to accomplish the goals of the program. One starts writing an object-oriented program by seeking objects relevant to the problem, studying their behavior and interactions, and sketching "blueprints" for the objects. Each blueprint is coded in C++ as a class.

9.2.1 Objects as Representations

A GPIB instrument control program manipulates objects representing each instrument. A class definition for objects that represent Acme model 130 voltage supplies might look like this:

ch9/SeeCommonality.C

```
class Acme130 {
public:
    Acme130(GPIBController_Stub& controller, int gpib_address);
    void set(float volts);
    double minimum() const;
    double maximum() const;
private:
    GPIBController_Stub my_controller;
    int my_gpib_address;
};
```

There is a constructor, Acme130(), and three other function members, set(), maximum(), and minimum(). Two data members are encapsulated, the my_controller object representing the GPIB hardware attached to the physical Acme 130 supply and the integer my_gpib_address representing the GPIB address.

The constructor is passed an integer that gives the GPIB address of a particular Acme 130 supply. The object constructed represents the corresponding supply. The constructor's code initializes the member data and connects to the controller:

ch9/SeeCommonality.C

```
Acme130::Acme130(GPIBController_Stub& controller, int gpib_address) :
    my_controller(controller),
    my_gpib_address(gpib_address) {
    my_controller.insert("Acme130", gpib_address);
}
```

In a real instrument control program other initialization actions might be required.

The set() member tests the voltage and then passes it, with the value of my_gpib_address to identify the physical voltage supply, to the GPIBController:

ch9/SeeCommonality.C

```
void Acme130::set(float volts) {
    if (volts > maximum() || volts < minimum()) throw "Acme 130 voltage out of range";
    my_controller.send(my_gpib_address, volts);
}
```

With our GPIBController_Stub version of GPIBController, set() just prints the voltage. The minimum() and maximum() functions return constants characteristic of the range limitations of the Acme 130 (see Exercise 9.1).

Consider executing the following code:

ch9/SeeCommonality.C

```
GPIBController_Stub gpib;
Acme130 volt_supply(gpib, 12);    // Supply at GPIB address 12.
volt_supply.set(3.6);
```

The first line establishes an object to represent the bus connected to the physical voltage supply. The second line causes memory to be allocated for the object state specified in the Acme130 class definition and invokes the Acme130() constructor to initialize the state. The third line runs the member function set() on the volt_supply object and with the value 3.6. The function's code sends 3.6 through the GPIB controller. The following output

```
(Acme130 now at address 12)
(GPIB instrument #12 sends value 3.6)
```

results from the functions defined in GPIBController_Stub on page 226.

We can now draw parallels between our Acme130 objects and the Acme 130 voltage supplies they represent. Physical objects can be characterized by their *behavior* and their *state*. A physical object's behavior is its response to stimuli; behavior is what an object can do. One behavior of a voltage supply might be: output voltage increases when the voltage select dial is rotated clockwise. A physical object's state is whatever information it contains; state is an object's generalized value. The voltage supply's state might be represented by the combination of its switch and dial settings. The state is whatever information is required to determine uniquely the object's response to stimuli.

Like physical objects, C++ objects have behavior in the form of their member functions. C++ objects also have state, either explicitly in the form of their member data or implicitly as external state accessed through computer peripheral devices like the GPIB interface.

The behavior of a physical object either changes its state, depends on its state, or both. A voltage supply produces a different voltage when its front-panel knob is rotated. Likewise a C++ object's behavior changes or depends on its state. The response to invoking set() with an Acme130 object both depends on and alters the object's state. It depends on the internal value of the GPIB address to route the voltage to the correct supply and it alters the supply's output, an implicit part of its state.

Describing C++ objects in terms of state and behavior rather than data and functions can sometimes make it easier to design object-oriented programs by providing a different vocabulary in which to think. It can also give us a different perspective on the goals of object design, as we now discuss.

9.2.2 State Representation Ambiguity and Encapsulation

Objects combine state and behavior. The representation of an object's state is potentially "ambiguous": The essential information content of an object can be represented in different but equivalent forms. A voltage supply's state could as well be represented by an indicator needle, currents on circuits inside the supply, or by the buttons pushed down on the front panel. The representations are different even though the essential information is identical.

The behavior of a physical object depends on its state: 3.6 volts appears across the output when the corresponding buttons are pushed. But different representations of that state yield identical behavior: Whether we choose to think about the indicator needle, the circuits, or the front panel buttons does not matter as long as we get 3.6 volts. The behavior, not the state representation, is the essential information.

As with physical objects, there are potentially many ways to represent a C++ object's state. Using set() on an Acme130 object to change the supply voltage does

not require knowing that the supply is connected via GPIB or knowing the GPIB address associated with the supply. Any code that uses only set() on an Acme130 need not be altered in any way if we later choose to upgrade our connection or switch to radio-controlled supplies.

The state and the behavior aspects of an object also interest us at different times. Designers of objects need to know details of state; users of objects concentrate on the behavior of objects, not on the details of their state. This is true for physical entities and for C++ objects. The Acme engineers must have precise blueprints for their supply, but how the supply might generate 3.6 volts is incidental to the user of an Acme 130 supply. The programmer who writes the Acme130 must know how the computer is attached to the supply, but the programmer who uses the supply does not care if it is attached by GPIB or radio control.

The different interests of designers and of users of objects result from the state-representation ambiguity of objects. Users are not directly interested in which representation is used for objects; designers must specify a specific representation.

Capitalizing on the fundamental concept of state-representation ambiguity for users of object behaviors is a central goal of object-oriented programming. Separating the code that specifies object implementations from the code that uses objects and allowing code that uses objects to access only behaviors—member functions—and not the state representation—member data—yields code that we can adapt and extend.

As we discussed in Section 8.3, preventing access to state representation is called information hiding. Effective information hiding requires two steps, one easy and one challenging. The easy part is using C++'s encapsulation mechanism:

- Declare member data private.

The difficult part is choosing a set of member functions that provide the full behavior of objects without exposing the state representation to using code. For example, if we write a member function for Acme130 called getAddress() that returns the value of my_gpib_address and another called setAddress() taking an int argument to reset the address, then the value of encapsulating my_gpib_address is diluted.

Information hiding allows us to exploit the possibility of having different state representations by picking the "best" representation according to criteria prevailing when we implement the object's behavior. If these conditions change, as when problems or computers or compilers or users change, the representation can be altered and the behaviors reimplemented without altering other parts of the program that only call the member functions. Changing our apparatus to use some other (non-GPIB) bus only requires changing the representation of Acme130; no code that sets voltages need be altered. Keeping the choice of state representation secret from client code, enabled by the binding of state and behavior, helps make programs adaptable.

If you write programs in terms of member function calls, eschewing direct access of member data, you can change the function implementations when you change the data representation, thus leaving the function calls of the program unaltered.

The advantages of encapsulation do not come for free. Calling functions to access data takes more compute time than directly accessing the data. Judicious use of inline functions can avoid this cost in critical cases.

9.3 Interfaces and Interface Categories

Programming with objects and classes gives us a new way to think about programming and helps structure our code. C++ also provides tools for coordinating groups of classes with common properties. We begin our exploration of such tools with groups we call *interface categories*. Classes in an interface category share certain specially marked functions called *virtual* functions. In this section, we motivate the use of interfaces when a group of classes shares common function specifications, and we describe the C++ virtual function mechanism.

9.3.1 Recognizing "is-usable-as" Commonality

Consider our simplified `Acme130` class representing voltage supplies attached to a GPIB system. Such a voltage supply might be a critical component in your experimental apparatus. Suppose lightning strikes and blows up your Acme 130. A colleague down the hall is not using her VoltOn (another fictitious brand) model 59 voltage supply and offers to loan it to you. You remove the Acme supply for repair and attach the VoltOn supply to your experiment and your GPIB bus; you remove the `Acme130` class from your program and replace it with a `VoltOn59` class. Now things slow down: Your program uses `Acme130` objects throughout. If you change them all to `VoltOn59` objects, you'll have to change them all back when you return the VoltOn supply.

Even though any one of a number of physical voltage supply objects will work in the physical apparatus, the program only works for one particular voltage supply programming object. We need to specify that an `Acme130` is usable as a `VoltageSupply` and a `VoltOn59` is usable as a `VoltageSupply`, and to write our program in terms of the common behavior of voltage supplies. `VoltageSupply` is an example of an interface category, a group of classes that share common function specifications.

To see how the aforementioned scenario might appear in code, consider a function for verifying the voltage from our supply. Suppose that, in addition to the Acme 130 voltage supply, we also had a computer-controlled voltmeter from the fictitious VoltyMetrics corporation. A class to represent these meters might be coded like this:

ch9/SeeCommonality.C

```
class VoltyMetrics {
public:
    VoltyMetrics(GPIBController_Stub& controller, int what_address);
    float read();
    // Other member functions go here.
private:
    GPIBController_Stub my_controller;
    int my_gpib_address;
};
```

Again the GPIB address is given to the constructor (definition not shown). The member function read() uses the address when communicating with the GPIB controller hardware to access the voltage value:

ch9/VoltyMetrics.h

```
float VoltyMetrics::read() {
    return my_controller.receive(my_gpib_address);
}
```

With the meter and supply objects defined, we can write a checkCalibration() function:

ch9/SeeCommonality.C

```
float checkCalibration(Acme130& supply, VoltyMetrics& meter, float tst_voltage) {
    // Relative error at specified test voltage.
    supply.set(tst_voltage);
    return abs(tst_voltage - meter.read()) / tst_voltage;
}
```

This code is easy to read, in part because the details of GPIB communications are hidden in the member functions for the supply and meter objects. This function can be called like this:

ch9/SeeCommonality.C

```
VoltyMetrics meter(gpib, 14);
Acme130   supply1(gpib, 12);
cout << "Acme130 relative error at 1 volt is: " <<
                                    checkCalibration(supply1, meter, 1.0) << endl;
```

Now we bring in our VoltOn59 class, which controls the loaner VoltOn supply:

ch9/SeeCommonality.C

```
class VoltOn59 {
public:
    VoltOn59(GPIBController_Stub& controller, int gpib_address);
    void set(float volts);
    double minimum() const;
    double maximum() const;
```

```
private:
    GPIBController_Stub my_controller;
    int my_gpib_address;
};
```

The member functions could be similar to the ones we wrote for Acme130 (see Exercise 9.2). We hit a snag as soon as we try to compile the calibration code: There is no checkCalibration() function for VoltOn59 supplies. The code

ch9/SeeCommonality.C

```
VoltyMetrics meter(gpib, 14);
VoltOn59 supply2(gpib, 17);
// WRONG:
cout << "VoltOn59 relative error at 1 volt is: " <<
                                    checkCalibration(supply2, meter, 1.0) << endl;
```

will not compile. The supply2 object is a VoltOn59 object, and the first argument of checkCalibration() requires a reference to an Acme130 object.

We could create a second checkCalibrate() function, one specifically for VoltOn59 objects. For small programs that will not last, this solution is adequate. As the number of cases begins to multiply, the amount of almost-but-not-quite identical code increases. Maintaining the collection of functions begins to consume our time, time better spent elsewhere.

The checkCalibration() function depends on a single function from Acme130, set(), for setting the test voltage. Encapsulation in the design of Acme130 has isolated the implementation of set() from the implementation of checkCalibration(): We could change Acme130 to use a different computer connection without altering checkCalibration(). The same encapsulation works in VoltOn59 and, moreover, VoltOn59 naturally provides the same function, set(). The common function set() is the only voltage supply function used by checkCalibration(). A member function that characterizes a common use for instances of a group of classes signals the potential application of virtual function interfaces.

9.3.2 Virtual Function Interfaces

We now introduce virtual functions, base classes, and calling functions through interface references. These C++ language components are needed to lift the set() function out of the particular Acme130 and VoltOn59 classes and into a more general and abstract VoltageSupply class. These mechanisms allow both Acme130 and VoltOn59 objects to be used as VoltageSupply objects and allow checkCalibration() to be written for any object usable as a VoltageSupply.

To generalize checkCalibration(), we first recognize that its purpose is to check calibration on voltage supplies, a task not related to Acme 130 supplies or to GPIB-connected supplies. To achieve its purpose, checkCalibration() uses a single function

related to voltage supplies, set(). The presence of this function characterizes all objects representing voltage supplies. We naturally expect this function in both Acme130 and VoltOn59. We want to abstract this commonality from Acme130 and VoltOn59.

Our plan is to express the common occurrence of set() in a new class, recode checkCalibration() to use this new class, and attach Acme130 and VoltOn59 to the new class. We call this new class an *interface base class*, or simply an *interface base*; we call the group of classes that are attached to an interface base an *interface category*, shorthand for the category of all classes sharing an interface base class.

The interface base represents the properties of voltage supplies in general, so we call it VoltageSupply. These properties of voltage supplies are member function declarations marked as abstractions by the keyword virtual:

ch9/VoltageSupply.h

```
class VoltageSupply {
public:
    virtual void set(float volts)       = 0;
    virtual float minimum()     const = 0;
    virtual float maximum()     const = 0;

    virtual ~VoltageSupply();
};
```

The declarations that end with =0 are *pure virtual functions*; no instances of a class with a pure virtual function can be created. As discussed further in Section 9.6, these pure virtual functions ensure that VoltageSupply is only used as an interface base. To ensure that objects can be deleted properly using interface base class pointers:

- Declare a virtual destructor in every interface base class unless it is derived from another interface base class that provides a virtual destructor.

This will be explained in more detail in Section 9.9.3. Such destructors must also be defined, typically to do nothing:

ch9/VoltageSupply.C

```
VoltageSupply::~VoltageSupply() { }
```

Next we revise the checkCalibration() function to work off the interface base class rather than the particular supply class. Functions like checkCalibration() that should work with voltage supplies in general can call the VoltageSupply virtual functions, and the actions that result will depend on the particular voltage supply object. The revised checkCalibration() takes not an Acme130 or VoltOn59 reference, but a VoltageSupply reference:

ch9/checkCalibration_VS.h

```
float checkCalibration(VoltageSupply& supply, VoltyMetrics& meter, float tst_voltage) {
    // Relative error at specified test voltage.
    supply.set(tst_voltage);
    return abs(tst_voltage – meter.read()) / tst_voltage;
}
```

Now let's suppose that we revise Acme130 and VoltOn59, in a manner described later, to put them in the VoltageSupply category and that we call the revised versions Acme130_VS and VoltOn59_VS, respectively. We can then call the same checkCalibration() function twice, once with an Acme130_VS object and once with a VoltOn59_VS object:

ch9/SeeVoltageSupply.C

```
VoltyMetrics  meter(gpib, 14);
Acme130_VS supply1(gpib, 12);
VoltOn59_VS supply2(gpib, 13);
cout << "Acme130_VS relative error at 1 volt is: " <<
                                        checkCalibration(supply1, meter, 1.0) << endl;
cout << "VoltOn59_VS relative error at 1 volt is: " <<
                                        checkCalibration(supply2, meter, 1.0) << endl;
```

In both cases, checkCalibration() calls the set() member of VoltageSupply, but in the first case the set() code for Acme130_VS objects is invoked and in the second case the set() code for VoltOn59_VS objects is invoked. This behavior results from the virtual specifier on the function set().

The final step in our plan is to revise the representations for physical supplies to put them in the VoltageSupply interface category. To declare that Acme130 will act like a VoltageSupply, we revise it to read

ch9/Acme130_VS.h

```
class Acme130_VS :
    public VoltageSupply {
public:
    Acme130_VS(GPIBController_Stub& controller, int gpib_address);
    virtual void set(float volts);
    virtual float minimum()  const;
    virtual float maximum() const;
private:
    GPIBController_Stub my_controller;
    int my_gpib_address;
};
```

This completes our abstraction of VoltageSupply commonality from the Acme130 class for use in functions like checkCalibration() that work for all supplies.

9.3.3 Derived Classes

Let's review the example of Section 9.3.2 to introduce the mechanics of base classes and virtual functions. Acme130_VS is said to be a *derived class*, derived from the VoltageSupply base class. In general, the terms derived class and base class refer to the relationship between two classes and not to the classes themselves. Later we shall see that Acme130_VS, for example, can also be a base class; conversely, VoltageSupply could have been derived from its own base classes.

A class can be derived from one or more base classes specified by a colon (:) following the class name, followed by a comma-separated list of base class specifications; each base class specification consists of one of the keywords public, protected, or private that is optionally followed by the keyword virtual, followed by the name of the base class. The base class must have been defined previously. We defer our introduction of the virtual specifier in front of a base class name until Section 10.5.

The public specifier means that an instance of the derived class can be used through references or pointers to the base class. This specifier allows the version of checkCalibration() that takes a VoltageSupply reference to be called with Acme130_VS objects. By "allows," we mean both that the code operates properly to call the appropriate set() function and that C++ type checks these calls. The meanings of private and protected specifiers for base classes are discussed in Chapter 10.

The VoltageSupply base class for Acme130_VS has a special form that we call an interface base class. We use this term to mean a base class designed to hold function declarations common among a group of classes. These common declarations represent properties abstracted from the classes. For example, the property of voltage setting is common to all voltage supply objects, and we abstracted from Acme130 and VoltOn59 a set() function to represent this property. Placing this declaration in the VoltageSupply class makes it a characteristic of all objects we use to represent voltage supplies, that is, of all objects derived from VoltageSupply.

You will not find the term *interface base class* in a C++ manual. Base classes are used in C++ programming for several purposes; we introduce the term to denote a base class used to specify virtual functions that are defined in derived classes and called through pointers or references to the base class. Generally, an interface base class is characterized by not having data members and having all member functions declared virtual. Base classes used for other purposes will be introduced in Chapters 11 and 10.

Now we return to the functions in the body of the derived class, switching for our example to the revised class VoltOn59_VS:

ch9/VoltOn59_VS.h

```
class VoltOn59_VS :
    public VoltageSupply {
```

```
public:
    VoltOn59_VS(GPIBController_Stub& controller, int gpib_address);
    virtual void set(float volts);
    virtual float minimum()  const;
    virtual float maximum() const;
private:
    GPIBController_Stub my_controller;
    int my_gpib_address;
};
```

Compare this class to VoltageSupply. Each of the virtual functions from VoltageSupply is declared here, without the = 0, meaning that this class supplies its own definition for each of these functions. Since all the pure virtual functions from VoltageSupply are defined, instances of VoltOn59_VS can be created.

The return type, argument types, and const specifier in the derived-class declarations for the virtual functions must be identical to those in VoltageSupply. Otherwise these functions will be considered to be different functions. (But see Notes and Comments 9.10.) C++ does not require respecifying virtual. But to make it clear that these functions are being defined for an interface, we recommend that you

■ Redeclare virtual functions as virtual in derived classes.

The definitions of the member functions of Acme130_VS and VoltOn59_VS are the same as in the original versions. The only changes to the class are the added VoltageSupply base class and the virtual function specifiers on the member function declarations. The revised classes have broader applicability—they "are usable as" voltage supplies—but otherwise they have the same behaviors as the original versions.

The revision was a recognition of common meaning. The set() functions in both Acme130_VS and VoltOn59_VS have the same purpose—as is signaled by the interface base class—but the implementation details are different—as is signaled by the virtual specifier on the interface base class function declaration. Code like checkCalibation() written for any class in the VoltageSupply category uses the interface base class, but each class in the category supplies a potentially different implementation of functions specified in the interface. We say that every class in the VoltageSupply category "is usable as" a VoltageSupply.

Overall, using an interface base class and virtual functions allowed the revised checkCalibration() to work for all objects representing voltage supplies rather than just objects representing Acme 130 voltage supplies or VoltOn 59 voltage supplies. In larger systems, this difference becomes vital to maintainable and extensible code.

9.4 Sketching Objects and Classes

Pictorial representations of programs can be useful for understanding their structure. Object-oriented programs have some advantage in visualization: The object view gives us something to draw. However, we can easily produce programs so complex that drawing them takes more time than it saves in understanding. Sketches serve primarily as a summary of some simple relationships, allowing us to think or talk about more complex ones. In this section, we introduce two kinds of sketches that we will use in the rest of the book: a sketch for objects and a common way of diagramming derivation relationships among classes. As we extend our C++ discussion in later chapters, we shall extend our diagrams.

9.4.1 Objects and Interfaces

Object diagrams illustrate data-member content and publicly visible function interfaces; we will extend them to indicate connections between functions and objects and among objects. Our object diagram for `Acme130_VS` appears in Fig. 9.1. The outer boundary of the diagram alludes to the object's finite size and its encapsulation of data. Inside the boundary, we sketch relevant member data, in this case an `int` object and a `GPIBController_Stub` object. Member data not relevant to the discussion at hand may be omitted. We draw built-in objects inside rectangles and class objects as arbitrary geometrical shapes. Member functions like `Acme130_VS`'s `set()` can be envisioned as conduits in the boundary, mediating requests for action on the encapsulated member data.

Along one edge of the boundary, the interface base class `VoltageSupply` is drawn as a narrow region with a name. This base class has no member data, only member function declarations. Hence it occupies negligible space but forms part of the mediating boundary. Although the word interface is synonymous with specifica-

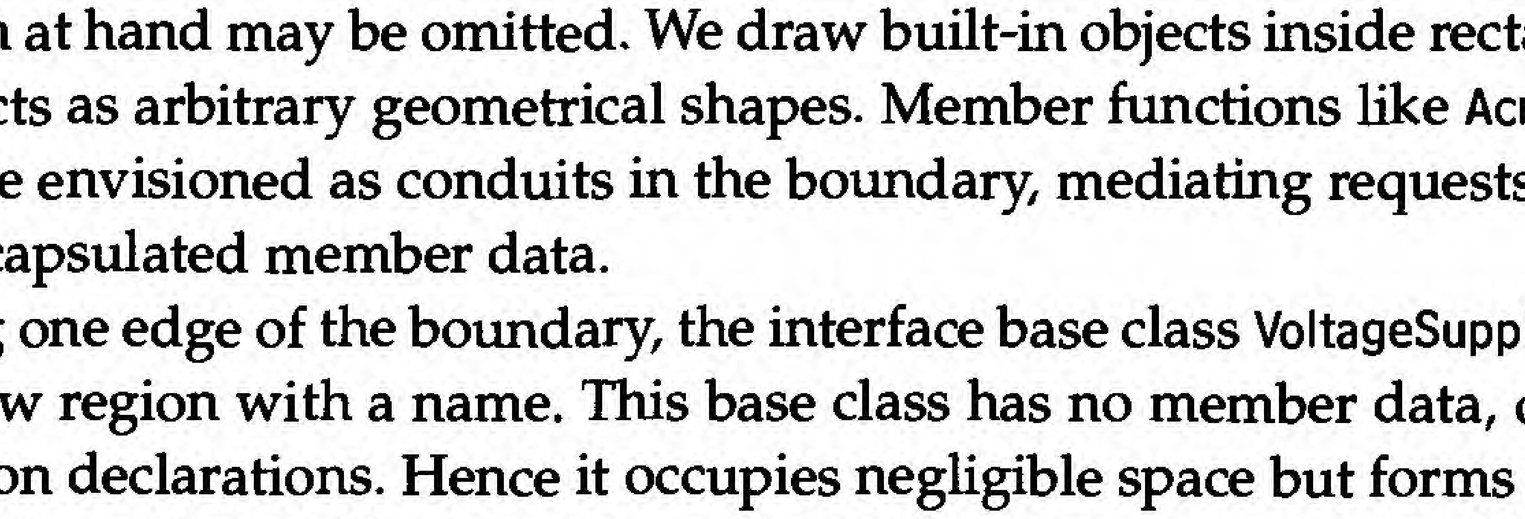
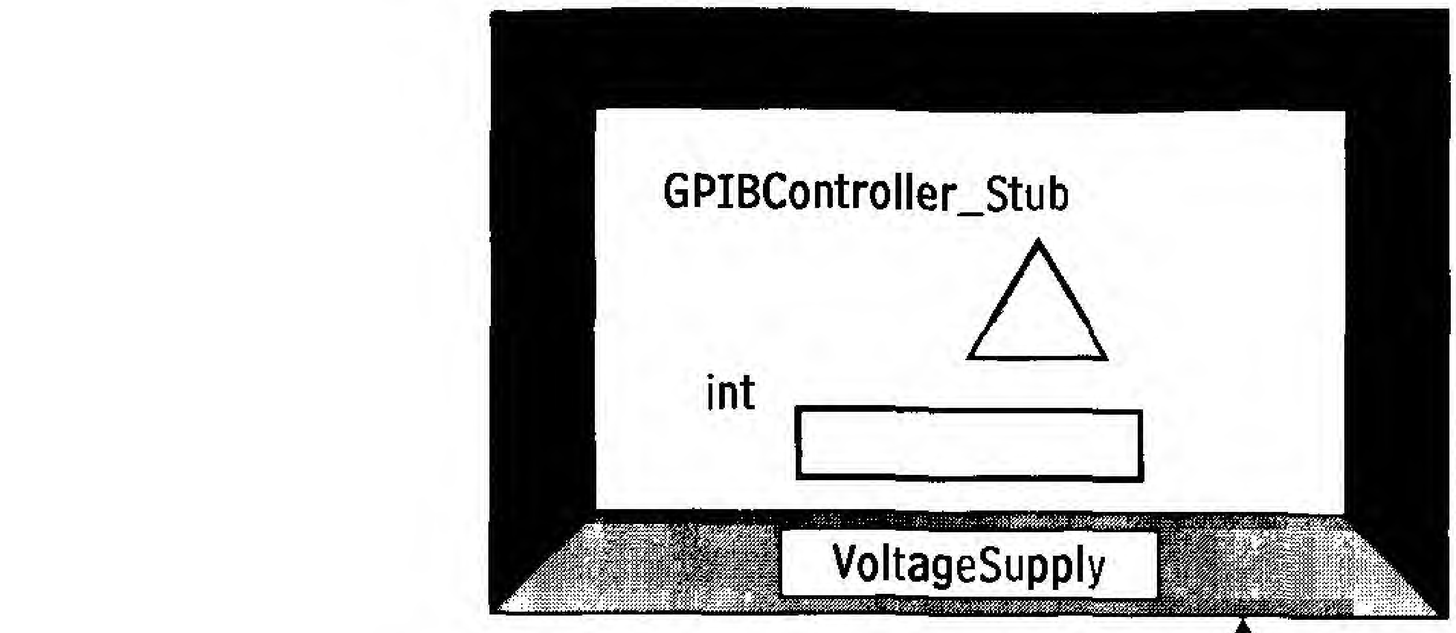


Figure 9.1 A Simple Object Sketch. Components are described in the text.




```
float checkCalibration(
    VoltageSupply& supply,
    VoltyMetrics& meter,
    float tst_voltage) {
  supply.set(tst_voltage);
  return abs(tst_voltage - meter.read()) / tst_voltage;
}
```

Figure 9.2 An Object Called from a Code Fragment.

tion when we speak of interface base classes, in our diagrams these base classes also appear on the interface between objects and the rest of the program.

Figure 9.2 shows the VoltOn59_VS object, this time to illustrate that we use arrows to indicate function calls. Functions are represented by their code rather than some sketch. Thus the call to VoltOn59_VS's set() function in checkCalibration() is represented by a line from the code to the object's interface. Note that the same VoltageSupply interface appears on the VoltOn59_VS object in this diagram and on the Acme130_VS object in Fig. 9.1. The checkCalibration() function calls through this interface, ignorant of the nature of the information on the object side of the interface.

9.4.2 Class DAGs

The relation induced among classes by derivation can be represented as a directed graph, as illustrated in Fig. 9.3. Classes are represented by their names, with each interface base class enclosed in a rectangle. An arrow is drawn from each derived class to its base class or classes. Such a diagram is called a *directed graph*. Since a class must be defined prior to use as a base class, it is not possible for the directed graph to have cycles (a path in the graph, paying attention to the arrows, that comes back to its starting position). A directed graph without cycles is called a *directed acyclic graph*, abbreviated *DAG*. Specifically, the diagram of classes and their derivation relation is called a *class DAG*. (Other names are also used;

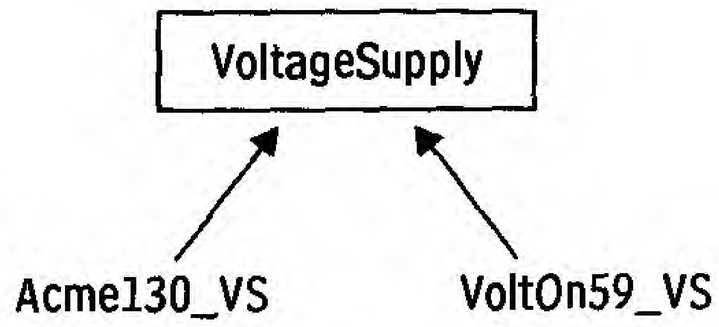


Figure 9.3 Class DAG for the VoltageSupply Classes Acme130_VS and VoltOn59_VS. The interface base class VoltageSupply is indicated by a box around the name.

see Notes and Comments 9.4.) The class DAG is a useful tool for understanding systems of classes.

9.5 Create Objects; Use Interfaces

Now we return to the virtual function mechanism and to the relation between objects and classes. The class DAG in Fig. 9.3 shows three classes, but only two kinds of objects were sketched in the previous section. No VoltageSupply objects were sketched and no VoltageSupply objects can be created. Nevertheless, as we have seen, we can use objects as VoltageSupplys. The meaning of this difference and the C++ mechanisms that provide this behavior are described in this section.

To understand the C++ virtual function mechanism, distinguish between the creation of objects—dependent on the details of a specific class—and the use of objects—dependent only on common function specifications in an interface. The following code creates an Acme130_VS object and then uses it as a VoltageSupply:

ch9/SeeVoltageSupply.¶

```
Acme130_VS supply1(gpib, 12);
VoltyMetrics meter(gpib, 14);
cout << "Acme130_VS relative error at 1 volt is: " <<
                                    checkCalibration(supply1, meter, 1.0) << endl;
```

The following code creates a VoltOn59_VS object and then uses it as a VoltageSupply:

ch9/SeeVoltageSupply.¢

```
VoltOn59_VS supply2(gpib, 13);
VoltyMetrics meter(gpib, 14);
cout << "VoltOn59_VS relative error at 1 volt is: " <<
                                    checkCalibration(supply2, meter, 1.0) << endl;
```

There is one checkCalibration() function, and it calls the set() function for a VoltageSupply. In the first case the VoltageSupply supply1 is an Acme130_VS and Acme130_VS's set() is called; in the second case the VoltageSupply supply2 is a VoltOn59_VS and VoltOn59_VS's set() is called.

Only VoltyMetrics, VoltOn59_VS, and Acme130_VS objects are involved. VoltageSupply appears only as a reference variable, never as a complete object. There are objects that are usable as voltage supplies, but no voltage supply objects. Every physical voltage supply is an object created according to a specific plan to meet the specifications expected of a voltage supply; every C++ object representing a supply is created from a specific class that provides definitions for the VoltageSupply interface base class virtual functions.

When checkCalibration() is called with an Acme130_VS object, the public derivation of Acme130_VS from the VoltageSupply base class allows C++ to set the VoltageSupply reference to refer to the Acme130_VS object. The function only uses the Acme130_VS object through a reference to the VoltageSupply interface base class of the object; or, as we sometimes say to be brief, the function only uses objects through their VoltageSupply interface.

Because one C++ language feature, the class, is used as both the common interface base class and the distinct, specific representation, the dramatic difference between the nature of classes like VoltageSupply on the one hand and classes like Acme130_VS and VoltOn59_VS on the other can be overlooked. To reinforce the differences, we emphasize that VoltageSupply is a means of specifying how all objects in a group of classes can be used, whereas Acme130_VS and VoltOn59_VS specify how objects can be created.

Creating an object ultimately requires knowing how its member data will be stored and initialized, information given by a specific class and its constructors: Creating an Acme130_VS requires invoking an Acme130_VS constructor, a function specific to Acme130_VS. Using an object only requires knowing that the object will act appropriately when some functions are called; using an Acme130_VS as a VoltageSupply only requires knowing that the object is an instance of a class derived from VoltageSupply, an interface base class with a set() function that sets the supply voltage.

VoltageSupply objects cannot be created; only references or pointers to the VoltageSupply interface of specific objects can be used. There is no VoltageSupply constructor and, in general,

- Interface base classes do not have constructors.

The interface describes only the abstraction; it does not have enough information to be a representation of a specific supply. The phrase "a VoltageSupply" can only mean "a reference or pointer to a VoltageSupply."

Conversely, an Acme130_VS object cannot be used as a VoltOn59_VS object and vice versa. These objects contain too much specific information to be substitutes: An Acme130_VS is connected to an Acme 130 supply and cannot be used as a VoltOn 59 supply.

The distinction between the information required to create and use objects forms the basis for abstraction and parallels directly the difference between creating and using physical objects. To order a voltage supply, we must specify completely the manufacturer and model, but to use a supply we need only know that it supplies voltage. Any one of a dozen different models manufactured by different companies would supply a voltage equally well. What's more, we don't care about the color of the voltage supply's case; we don't care if the voltage supply also reads voltage; we don't care if the voltage supply plays the radio; we only care that it can supply a voltage. As a user we seek an object that fulfills the abstraction "voltage supply": That is what we mean by the phrase "a voltage supply."

9.6 Interface Base Classes

As we have mentioned before, C++ has base classes, but not interface base classes; interface base classes are a usage of base classes. This section characterizes the usage.

C++ allows member data in base classes. However, recognize that an interface specifies behavior, not state:

- Interface base classes should have no member data.

Member data in an interface base would imply that the state of the interface, rather than the state of the object, controls some actions of the member functions. This control will limit your ability to implement the interface in new ways. Chapter 10 discusses effective ways to use base classes with member data, base classes that do not correspond to interfaces.

Since interface bases do not provide state, they usually do not provide definitions for the functions they specify. The principal exception is a member function that defines a relationship or constraint among the properties of the abstraction the interface base specifies. For example, we might choose to define a reset() function in VoltageSupply that is defined as the statement set(minimum()). Then all VoltageSupply objects have the property that reset() means "set yourself to your minimum voltage."

- Functions defined by an interface base should be defined solely in terms of other member functions specified by the interface base.

Except for these constraint functions, virtual members of interface base classes are declared to specify an interface, not to provide function implementations. Therefore,

- Declare interface base member functions to be pure virtual functions.

No function body need be provided for these functions. The inability to supply a meaningful definition for such functions is a logical consequence of abstraction. We know that all voltage supplies controlled via computers must have a set() function to set the voltage, but every different supply has a different way to set that voltage: We can declare set() in VoltageSupply, but we cannot define it.

A base class with at least one pure virtual function is called an *abstract base class*, and C++ does not allow instances of abstract base classes to be created. VoltageSupply is an abstract base class, and C++ prohibits creating instances of it. Both Acme130_VS and VoltOn59_VS define set(), but had we forgotten to define one (say, in VoltOn59_VS), the C++ compiler would not compile code that attempts to create a VoltOn59_VS object, forcing us to go back and supply the function. These consistency checks lead us to suggest that

- Interface base classes should be abstract base classes.

Keep in mind that we use the term interface base class to indicate a base class written to be used as an interface. You can use any base class having virtual functions—whether pure or not—as an interface base class. However if your interface base class is not an abstract base class, the compiler will not flag noninterface uses of the base class. Using abstract base classes as interface base classes puts the compiler to work helping to maintain the intended abstraction.

9.7 Multiple Interfaces

A class may belong to several interface categories; that is, it may be derived from several interface bases. For example, consider that our Acme 130 voltage supply is connected via the GPIB. Other instruments, some that are not voltage supplies, would also be connected via GPIB. Operations on these instruments related to the GPIB, like sending and receiving data, are abstract operations with distinct implementations according to the type of instrument. Functions might be written using only these operations, and there is no reason to rewrite such functions for every GPIB instrument.

For example, consider a function that reads data from one GPIB instrument and sends the result to another one, possibly located on a different GPIB:

ch9/tGPIBInstrument.C

```
void transferOnGPIB(GPIBInstrument& from, GPIBInstrument& to) {
    to.send( from.receive() );
}
```

Here we have made up an interface base class, GPIBInstrument, specifying the common properties of GPIB instruments:

ch9/GPIBInstrument.h

```
class GPIBInstrument {
public:
    virtual void send(const char*) = 0;    // Command.
    virtual void send(float f)     = 0;    // Command with value.
    virtual float receive()        = 0;    // Data point.

    virtual ~GPIBInstrument();
};
```

To allow Acme130_VS to work in the transferOnGPIB() function, and more generally for Acme130_VS to be usable as a GPIBInstrument, we must derive it from this interface base:

ch9/Acme130_VS_GI.h

```
class Acme130_VS_GI :
    public VoltageSupply,
    public GPIBInstrument {
public:
    Acme130_VS_GI(GPIBController_Stub& controller, int gpib_address);

// VoltageSupply interface
    virtual void set(float volts);
    virtual float minimum() const;
    virtual float maximum() const;

// GPIBInstrument interface
    virtual void send(const char*);    // Command.
    virtual void send(float f);        // Command with value.
    virtual float receive();           // Data point.

private:
    GPIBController_Stub my_controller;
    int my_gpib_address;
};
```

In this revision, the second base class appears on a comma-separated list following the colon. We add _GI to the name to mean the version with the GPIBInstrument interface. The public specifier, allowing references to GPIBInstruments to be created for Acme130_VS_GI objects, is the same as we described for VoltageSupply in Section 9.3.3.

The new GPIBInstrument functions can be defined in terms of the my_controller and my_gpib_address members:

ch9/Acme130_VS_GI.C

```
void Acme130_VS_GI::send(const char* cmd){
    my_controller.send(my_gpib_address, cmd);
}
```

The others are defined similarly.

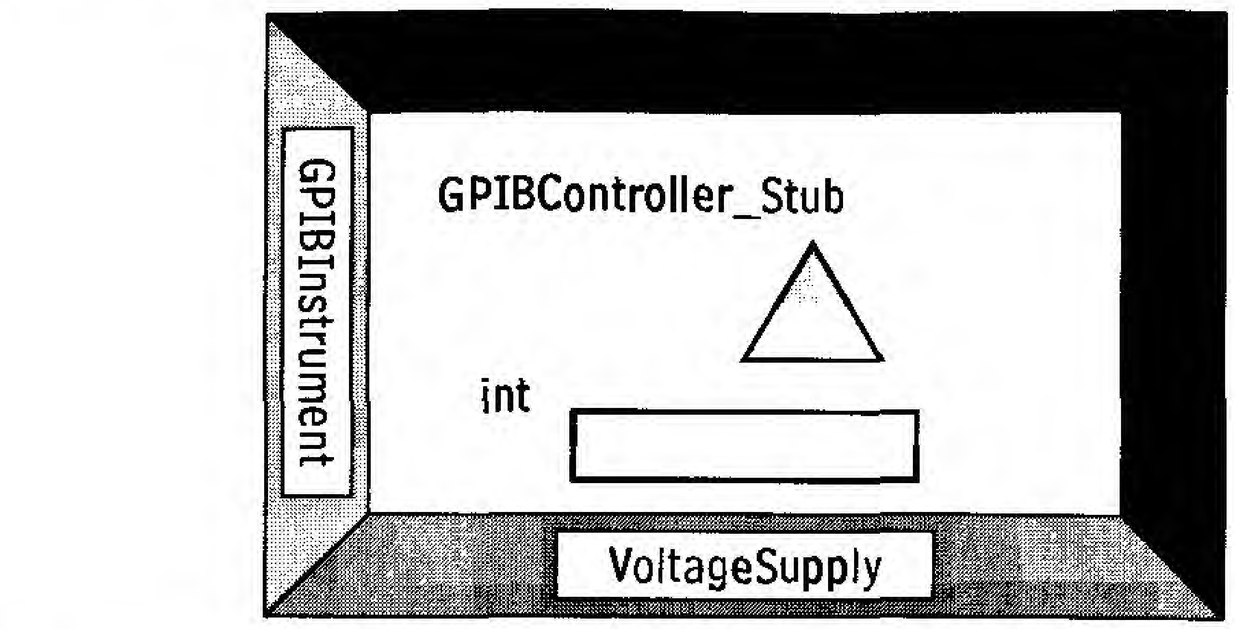


Figure 9.4 Object Sketch for an Object Representing an Acme 130 Voltage Supply Using Two Interface Bases.

The VoltageSupply member functions for this new class need not be changed. However, if desired, they can be altered to use the new GPIBInstrument member functions:

ch9/Acme130_VS_GI.C

```
void Acme130_VS_GI::set(float voltage){
    if (voltage > maximum() || voltage < minimum()) throw "Acme 130 voltage out of range";
    send(voltage);
}
```

Most important, the checkCalibration() function and all similar functions written for VoltageSupply references need not be rewritten or even recompiled.

Figure 9.4 sketches this version of our Acme 130 class. Two "sides" of the object now have interfaces: Functions can be written to view this object through references to its VoltageSupply interface, its GPIBInstrument interface, or directly as an Acme130_VS_GI object.

Figure 9.5 sketches the class DAG for this version of Acme130_VS_GI, including a version of VoltOn59 with GPIBInstrument (Exercise 9.3), and a version of VoltyMetrics redesigned to use Voltmeter and GPIBInstrument interfaces (Exercise 9.4). Each class shown is derived from two interface bases, as indicated by two arrows from each class.

The multiple interfaces we have developed here are independent: Being a voltage supply is completely independent of being a GPIB-attached instrument. Multiple interfaces also arise when one category of classes builds on the common specification of another category. Chapter 13 describes an extended example of multiple interfaces used in that way.

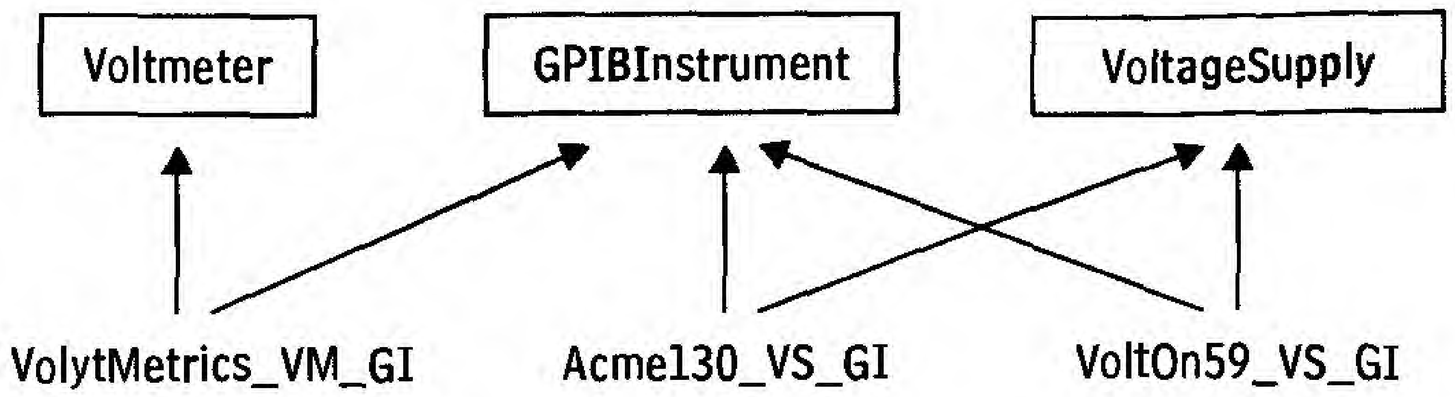


Figure 9.5 Class DAG for Classes Derived from Multiple Interfaces.

9.8 Using Interfaces as Components

Thus far, we have described virtual function calls from functions that take references or pointers to interface base classes. Such functions using the interface are called *client functions* of the interface. The checkCalibration() function is a client function of VoltageSupply; the transferOnGPIB() function is a client function of GPIBInstrument.

Classes can also be clients of an interface. This means that an instance of the class stores a reference or pointer to the interface. The interface reference or pointer might be stored in the object as a member datum or as an element in a collection of interface references or pointers. We give examples of each kind in this section.

9.8.1 Interface Reference Members

Each revision of our Acme130 class has made it usable through an additional interface. We now look inside the latest revision, Acme130_VS_GI, and observe that it uses a fixed GPIB controller, GPIBController_Stub, not any object usable as a GPIB controller. Although a physical Acme 130 could be connected via a GPIB bus cable to any of several brands of GPIB controller boards, our representation only works with one particular controller.

To work with any object usable as as a GPIB controller, we specify an interface base:

ch9/GPIBController.]

```
class GPIBController {
public:
    virtual void insert(const char* device_name, unsigned int address) = 0;
    virtual void send(unsigned int address, const char* cmd) = 0;
    virtual void send(unsigned int address, float f) = 0;
    virtual float receive(unsigned int address) = 0;

    virtual ~GPIBController();
};
```

Then we can make a version of our GPIB controller that is in the corresponding interface category like this:

ch9/GPIBController_GC.h

```
class GPIBController_GC :
    public GPIBController {
public:
    virtual void insert(const char* device_name, unsigned int address) {
        cout << "(" << device_name << " now at address " << address << ")" << endl;
    }
    virtual void send(unsigned int address, const char* cmd) {
        cout << "(" << "GPIB instrument #" << address <<
                                                    " sends " << cmd << ")" << endl;
    }
    virtual void send(unsigned int address, float f) {
        cout << "(" << "GPIB instrument #" << address <<
                                                    " sends value " << f << ")" << endl;
    }
    virtual float receive(unsigned int address) {
        cout << "(" << "Please enter number for GPIB instrument #" << address << ": ";
        float f;
        cin >> f;
        cout << ")" << endl;
        return f;
    }
};
```

Compare this version to the GPIBController_Stub class on page 226. The interface base class has been added and the functions have been declared virtual.

A version of Acme130 that works with any controller represented by an object in the GPIBController interface category can now be written:

ch9/Acme130_VS_GI_GC.h

```
class Acme130_VS_GI_GC :
    public VoltageSupply,
    public GPIBInstrument {
public:
    Acme130_VS_GI_GC(GPIBController& controller, int gpib_address);

    // ... member functions, as in Acme130_VS_GI ...

private:
    GPIBController& my_controller;
    int my_gpib_address;
};
```

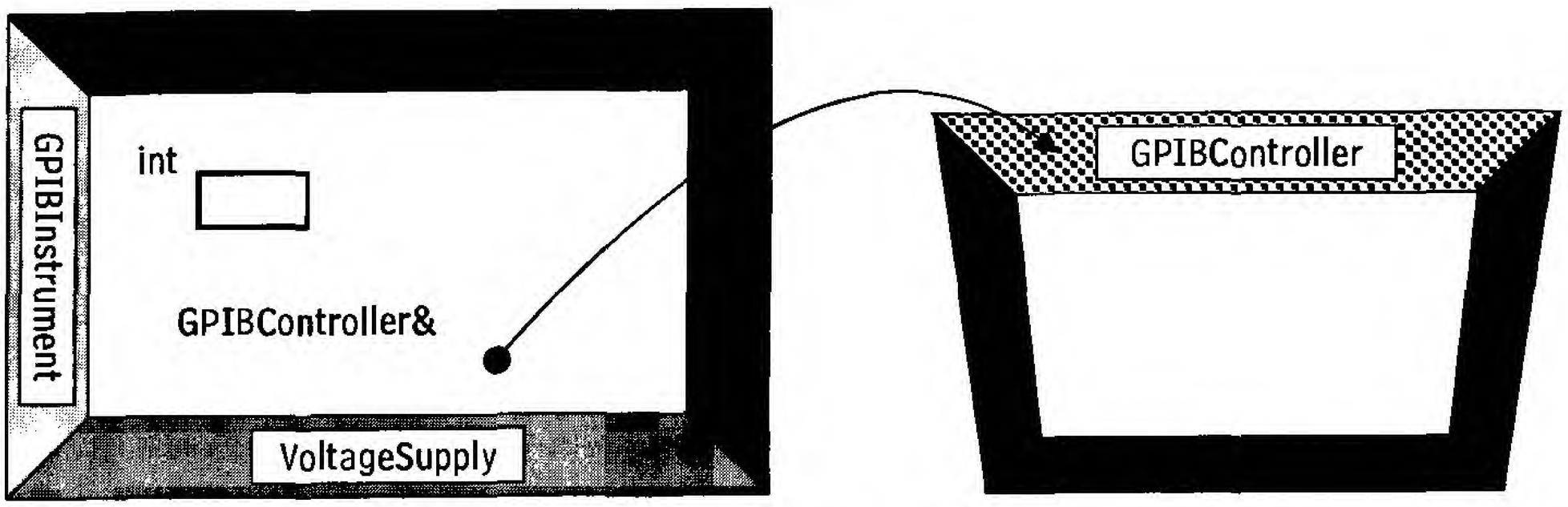


Figure 9.6 Object Sketch Showing an Object That Holds a Reference to an Interface.

The only changes we made were to have the constructor take a reference to the interface, instead of to a specific object, and to use a reference data member to hold that interface reference instead of holding a particular type of controller.

The reference is initialized by the constructor:

ch9/Acme130_VS_GI_GC.C

```
Acme130_VS_GI_GC::Acme130_VS_GI_GC(GPIBController& controller, int gpib_address) :
    my_controller(controller),
    my_gpib_address(gpib_address) {
    my_controller.insert("Acme130_VS_GI_GC", gpib_address);
}
```

The revised object is sketched in Fig. 9.6. As illustrated, the only knowledge we have about the controller object is that it provides a GPIBController interface. Controllers from various manufacturers can be accommodated by deriving from the GPIBController base class and implementing the virtual functions with calls specific to the particular hardware.

9.8.2 Coordinating Interface Members

As another example of an object containing interface references, consider representing a simple apparatus for measuring a current-voltage curve. Various electronic components like diodes can be characterized by measuring the current flowing through the component as a function of the voltage applied across it. An automated current-voltage or IV (I for electric current, V for voltage) tester could be created using a computer-attached voltage supply and voltmeter. Notice that nothing in this description requires a particular supply or meter: An IVTester object should work for any instance of a class in the VoltageSupply category and any instance of a class in a Voltmeter category that represents common properties of voltmeters. (See Exercise 9.4.)

To create an IVTester object using the abstract interfaces rather than being tied to a specific supply, we write it with reference member data:

ch9/IVTester.h

```
class IVTester {
public:
    IVTester(VoltageSupply& vs, Voltmeter& vm);
    double current(double voltage);
private:
    VoltageSupply& the_voltage_supply;
    Voltmeter& the_voltmeter;
};
```

The constructor initializes the reference members:

ch9/IVTester.C

```
IVTester::IVTester(VoltageSupply& vs, Voltmeter& vm) :
    the_voltage_supply(vs),
    the_voltmeter(vm) {
}
```

The current() function that measures one point on the IV curve calls virtual functions through these stored references:

ch9/IVTester.C

```
double IVTester::current(double voltage) {
    the_voltage_supply.set(voltage);
    return the_voltmeter.read();
}
```

Of course this function is trivial; a more realistic function would, for example, check the voltage against the supply's range and delay for slew time in the supply before reading the meter.

The IVTester will operate for any pair of VoltageSupply and Voltmeter objects we supply to the constructor. For example, the following code runs through a 10-volt IV curve:

ch9/tIVTester.C

```
GPIBController_GC gpib;
Acme130_VS_GI_GC supply(gpib, 12);
VoltyMetrics_VM_GI meter(gpib, 13);
IVTester iv(supply, meter);
double v_step = 1.0;
for (int i = 0; i < 10; i++) {
    cout << iv.current(v_step * i) << endl;
}
```

Each call to current() for the IVTester causes a call to set() for the VoltageSupply. The actual function called depends on the object referred to by the member datum the_voltage_supply. In this case, Acme130_VS_GI_GC's set() is called.

If we replace the Acme 130 object in the preceding example with a VoltOn 59 object, our IVTester works just fine. The IVTester only uses a VoltageSupply reference, it does not care how the interface referenced is implemented.

In comparing this use of interfaces to the checkCalibration() use, consider the connection between checkCalibration(), the client function that uses the interface, and supply1, the object with the interface. The connection begins at the function call when the reference is bound to the object; it ends when the function returns.

For classes, the connection between an IVTester object and the supply and meter objects lasts until the IVTester is no longer needed. Many different operations through the VoltageSupply and Voltmeter interfaces can be performed via the IVTester during its lifetime. For this reason, client classes can express more sophisticated uses of objects through interfaces than can client functions.

The IVTester object depends on the particular supply and meter only in the constructor call, where the interface references attach to the actual objects. Thus IVTester is itself an abstraction, a representation of the concept of a current-voltage tester. It is more general than any similar class based on specific voltage supplies and meters; moreover, its operation is completely independent of the GPIB connection between the computer and the apparatus. This generality or abstraction allows IVTester to continue to be useful through changes to the implementation of the voltage supply and meter.

Both client classes and client functions can be written in terms of pointers to interface base classes rather than using references to interface base classes, as we have done in our examples thus far. For the operations we have described here, only the notation of function calls would change. More significant differences are covered in Chapter 14. Our next example uses pointers to interfaces.

9.9 Creating and Using Arrays of Interfaces

Collections of interfaces to objects go beyond the single member datum used in the preceding section to allow collective and delayed action on groups of objects related only by some common properties. In this section, we illustrate this idea by building an array of pointers to GPIBInstrument objects. The array might contain pointers to voltage supplies, voltmeters, or other instruments not yet invented. As long as they are objects in the category of classes derived from GPIBInstrument, they can be controlled via the array pointers.

For example, consider a simulator for a GPIB board in a computer. The real board comes with software that allows commands to be sent to and data received from instruments on the bus. Our simulator will accept commands and return

data by routing the send() and receive() commands to simulated instruments. Each simulated instrument must respond to send and receive commands in ways specific to each instrument. Again, the pattern of a common specification with distinct implementation leads us to virtual functions.

9.9.1 Simulating Instruments and Experiments

We will call the simulator for the Acme 130 voltage supply an Acme130Simulation and the simulator for the VoltyMetrics voltmeter a VoltyMetricsSimulation. They will be contained in a simulation of the GPIB controller board. When we issue set() on an Acme130 object, the voltage value must be sent by the controller board simulation to the corresponding Acme130Simulation object; when we issue a read() call on a VoltyMetrics object, the value from the corresponding VoltyMetricsSimulation object must be sent back by the controller board object. The communications must be accomplished using the GPIBController member functions (page 246) only.

We call a class able to participate in the simulation a GPIBInstrumentSimulation and specify it like this:

ch9/GPIBInstrumentSimulation.h

```
class GPIBInstrumentSimulation {
public:
    virtual void send(const char*) = 0;
    virtual void send(float f) = 0;
    virtual float receive() = 0;

    virtual ~GPIBInstrumentSimulation();
};
```

This expresses the common function interface but allows different implementations.

Our implementation of Acme130Simulation derives from the interface base class and redeclares the virtual functions to show that it will define them:

ch9/Acme130Simulation.h

```
class Acme130Simulation :
    public GPIBInstrumentSimulation {
public:
    Acme130Simulation(ExperimentSimulation& e);
    virtual void send(const char*);
    virtual void send(float);
    virtual float receive();
private:
    ExperimentSimulation& the_experiment;
};
```

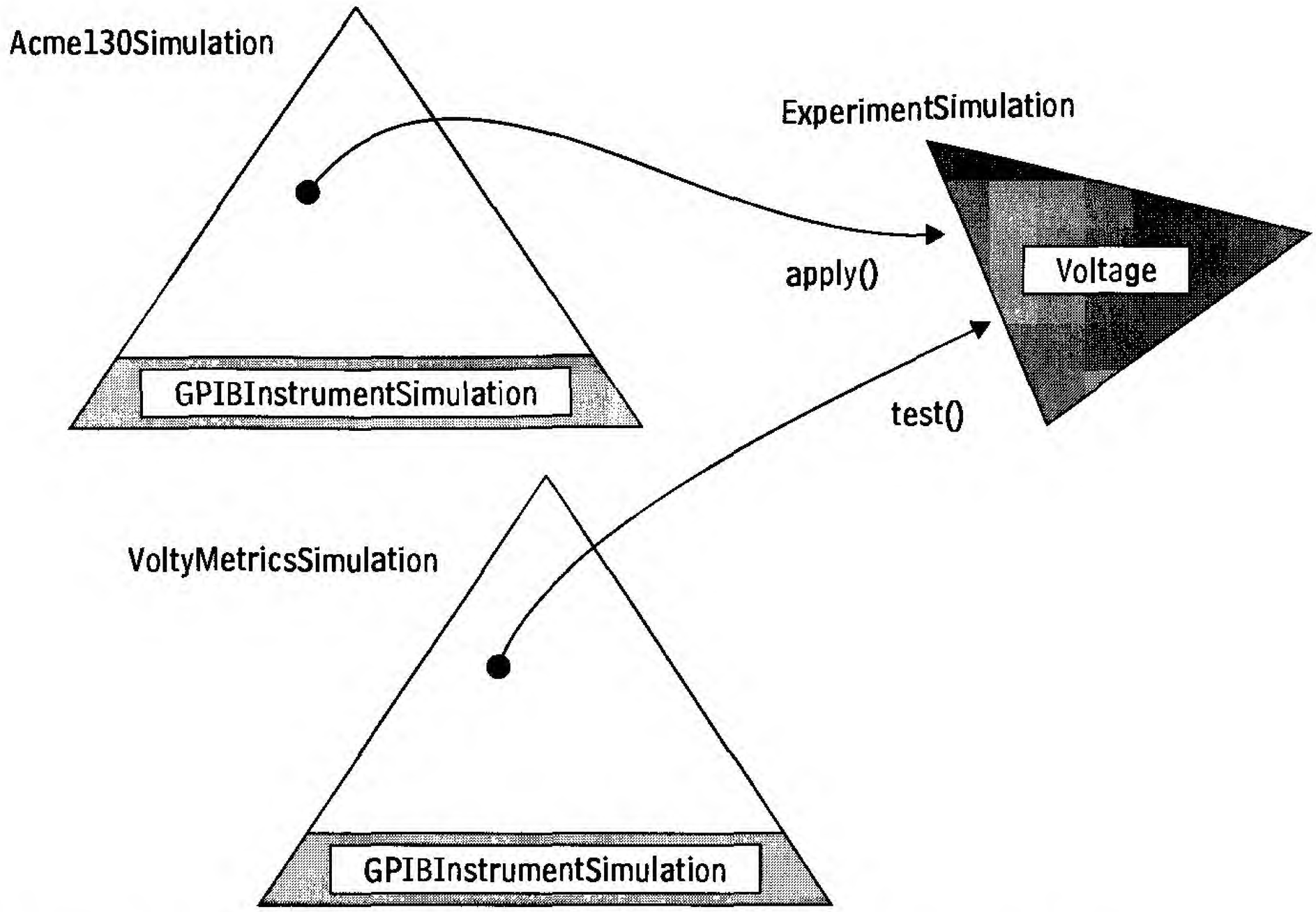


Figure 9.7 Sketch of Objects Used to Simulate a Current-Voltage Measurement.

This class contains a reference to a shared ExperimentSimulation object designed to simulate the state and behavior of the experimental apparatus. As shown in Fig. 9.7, a current-voltage measurement can be simulated by coupling a VoltyMetricsSimulation to the same ExperimentSimulation object:

ch9/VoltyMetricsSimulation.l

```
class VoltyMetricsSimulation :
    public GPIBInstrumentSimulation {
public:
    VoltyMetricsSimulation(ExperimentSimulation& e);
    virtual void send(const char*);
    virtual void send(float f);
    virtual float receive();
private:
    ExperimentSimulation& the_experiment;
};
```

The ExperimentSimulation object shared between the supply and the meter holds a voltage value:

ch9/ExperimentSimulation.l

```
class ExperimentSimulation {
public:
    void   apply(double voltage);   // Called by voltage supply simulation.
    double test();                  // Called by voltmeter simulation.
```

```
private:
    double the_voltage;
};
```

The apply() function, to be called by the Acme130Simulation send(),

ch9/Acme130Simulation.C

```
void Acme130Simulation::send(float f) {
    the_experiment.apply(f);
}
```

might just set the the_voltage member, whereas the test() member, to be called by the meter simulator,

ch9/VoltyMetricsSimulation.C

```
float VoltyMetricsSimulation::receive() {
    return the_experiment.test();
}
```

might return some function evaluated at the_voltage. See Exercise 9.6.

With some classes in the GPIBInstrumentSimulation category, we can return to the simulation of the GPIB controller itself. The declarations for the member functions of the simulated GPIB controller must match the GPIBController interface specified on page 246. The controller must keep track of the simulator for each GPIB instrument registered with it so that later send() and receive() commands can be routed to the correct simulator indicated by the address argument. Since the GPIB addresses must be between 0 and 31, an array can hold the simulators. Of course neither an array of Acme130Simulation objects nor an array of VoltyMetricsSimulation objects will do: We need an array that holds simulators independent of the kind of object they simulate. An array of pointers to GPIBInstrumentSimulation interfaces matches our needs exactly.

9.9.2 An Object Factory

The controller simulator must also be able to "attach" an appropriate simulator to the simulated bus when the simulator's insert() member is called. When insert() is called with a real controller, the controller checks that the specified device is attached physically to the GPIB at the specified address. When insert() is called with a simulated controller, some way is needed to create a simulator appropriate for the device being inserted. We solve this problem with a simulator "factory" interface:

ch9/SimulatorFactory.h

```
class SimulatorFactory {
public:
    virtual GPIBInstrumentSimulation*
        create(const char* device_name, ExperimentSimulation& exp) const = 0;
    virtual ~SimulatorFactory();
};
```

A class in this category must provide a create() function that, given a GPIB device name and an ExperimentSimulation, creates an appropriate simulator and returns a pointer to it. We'll see an example after looking at how the controller simulator uses the "factory."

The controller simulator looks like this:

ch9/GPIBController_GIS

```
class GPIBController_GIS :
    public GPIBController {
public:
    GPIBController_GIS(const SimulatorFactory& factory);
    virtual ~GPIBController_GIS();

// GPIBController interface
    virtual void insert(const char* device_name, unsigned int address);
    virtual void send(unsigned int address, const char* buf);
    virtual void send(unsigned int address, float f);
    virtual float receive(unsigned int address);
private:
    const SimulatorFactory& simulator_factory;
    CheckedSimpleArray<GPIBInstrumentSimulation*> simulators;
    ExperimentSimulation the_experiment;
};
```

Inside we hold a reference to the simulator factory interface base class, an array of pointers to the simulators created thus far, and the experiment that ties the simulated instruments together.

The constructor takes a reference to a SimulatorFactory to use for creating simulators. This reference is used to initialize the simulator_factory member:

ch9/GPIBController_GIS.C

```
GPIBController_GIS::GPIBController_GIS(const SimulatorFactory& factory) :
    simulator_factory(factory),
    simulators(31) {
    simulators = 0;
}
```

The constructor also sets up the array of pointers to simulators, initializing the pointers to the null pointer.

The insert() function sets values into the array, using the simulator "factory" to create the appropriate simulator:

ch9/GPIBController_GIS.C

```
void GPIBController_GIS::insert(const char* device_name, unsigned int address) {
    simulators[address] = simulator_factory.create(device_name, the_experiment);
}
```

Notice that this function connects a particular simulator to an entry in the array that corresponds to the GPIB address of the simulated instrument, but the choice of simulator is made by the "factory."

The send() and receive() functions each obtain a pointer to the simulator for the device at a specified address, check that a simulator has been inserted at that address, and invoke the corresponding function in the simulator:

ch9/GPIBController_GIS.C

```
void GPIBController_GIS::send(unsigned int address, const char* msg) {
    GPIBInstrumentSimulation* ip = simulators[address];
    if (ip != 0) ip -> send(msg);
    else throw "No simulator at specified GPIB address";
}

void GPIBController_GIS::send(unsigned int address, float f) {
    GPIBInstrumentSimulation* ip = simulators[address];
    if (ip != 0) ip -> send(f);
    else throw "No simulator at specified GPIB address";
}

float GPIBController_GIS::receive(unsigned int address) {
    float val;
    GPIBInstrumentSimulation* ip = simulators[address];
    if (ip != 0) val = ip -> receive();
    else throw "No simulator at specified GPIB address";
    return val;
}
```

Notice that these functions only see GPIBInstrumentSimulation pointers and they cannot tell—nor do they need to tell—which simulator they are operating on.

Now let's try a simulation. The first step is to supply a "factory" class in the SimulatorFactory category:

ch9/tSimulator.C

```
class MySimulators :
    public SimulatorFactory {
public:
    virtual GPIBInstrumentSimulation*
        create(const char* device_name, ExperimentSimulation& exp) const;
};
```

The create() function compares the device name specified against a list of devices it knows how to simulate and either creates a simulator or throws an exception:

ch9/tSimulator.C

```
GPIBInstrumentSimulation*
MySimulators::create(const char* device_name, ExperimentSimulation& exp) const {
    GPIBInstrumentSimulation* sim_p;
    if (strcmp(device_name, "Acme130_VS_GI_GC") == 0) {
        sim_p = new Acme130Simulation(exp);
    }
    else if (strcmp(device_name, "VoltyMetrics_VM_GI") == 0) {
        sim_p = new VoltyMetricsSimulation(exp);
    }
    else throw "Simulator for specified device not available";
    return sim_p;
}
```

(The strcmp() function is available via the ANSI C header file string.h; it compares two character strings, returning zero when they are identical.)

The simulated controller is initialized like this:

ch9/tSimulator.C

```
MySimulators simulators;
GPIBController_GIS gpib(simulators);
```

The code for the simulation itself is simply the code we want to simulate:

ch9/tSimulator.C

```
Acme130_VS_GI_GC supply(gpib, 12);
VoltyMetrics_VM_GI meter(gpib, 13);

IVTester iv(supply, meter);
double v_step = 1.0;
for (int i = 0; i < 10; i++) {
    cout << iv.current(v_step * i) << endl;
}
```

Compare this with the analogous code on page 249. When the Acme130_VS_GI_GC and VoltyMetrics_VM_GI objects are created, the corresponding simulators are also created by the GPIBController_GIS object; when the IVTester runs, the simulator is exercised. For example, if we use an ExperimentSimulation with a test() function that just returns the value set by the supply ($x = y$ curve), the code simulation code prints the integers from zero to nine.

9.9.3 Deleting Arrays of Interfaces

One detail remains: Simulators are created using new but never deleted. The simulated controller's destructor takes care of this:

ch9/GPIBController_GIS.C

```
GPIBController_GIS::~GPIBController_GIS() {
    for (int i = 0; i < simulators.numElts(); i++) {
        delete simulators[i];
    }
}
```

This code exploits the fact that invoking the delete operator on a null pointer has no effect: There is no need to check that there is actually a simulator at an address before deleting it.

The destructor code also illustrates why an interface base must have a virtual destructor member function. The simulators array contains pointers to the interface base, GPIBInstrumentSimulation, but we are actually deleting an Acme130Simulation or VoltyMetricsSimulation object. When an object is deleted, C++ runs the object's destructor: Given that we only have a pointer to the interface base, which destructor should C++ run, ~Acme130Simulation() or ~VoltyMetricsSimulation()? The answer depends on which kind of object the pointer points to, just as the action to be taken by calling send() depends on the object pointed to. Such behavior is exactly what declaring a member to be virtual arranges. Since one can't predict when writing an interface base class whether any class eventually derived from the base will have a destructor member, it is safest to anticipate the possibility by providing a virtual destructor in the interface base.

9.10 Exceptions and Interfaces

We introduced the use of exceptions in Section 4.5. The key idea is that an object can be thrown by a throw statement and caught by a catch clause whose formal argument type matches the type of the thrown object. This scenario is yet another example of the distinction between creating and using an object: The object thrown is created in the throw statement but used in the catch clause. Effective exception management builds on this dual creator/user view:

- Create specific error-handling objects; handle them through interfaces.

- Recognize the common features of errors and the procedures to handle them; put those common features into interface categories.

In much of our code we use exception classes that share one behavior: They can provide a class-specific error message. This common behavior becomes a pure virtual message() function in SciEngErr:

SciEng/SciEngErr.h

```
class SciEngErr {
public:
    virtual String message() const = 0;
    virtual ~SciEngErr();
};

ostream& operator<<(ostream& s, const SciEngErr& e);
```

(The String class provides simple operations on character strings, including the ability to use the >> and << operators with a String object as source or destination, respectively. String is developed in Section 18.2.) The message produced by any exception class can be written to an output stream using the output operator:

SciEng/SciEngErr.C

```
ostream& operator<<(ostream& s, const SciEngErr& e) {
    return s << e.message();
}
```

Let's see how this interface could be used with the CheckedSimpleArray<T> class of Section 4.5. When an error is detected, one of the exception objects SubscriptRangeError or ArraySizeError is thrown and, as illustrated by the code on page 109, there is a catch clause to handle each kind of exception object. But if all that we're going to do is print an error message, as is often the case, we don't care which object was thrown, as long as it can provide a message. So we revise the exception classes to conform to the SciEngErr interface:

ch9/CheckedSimpleArray.h

```
class SubscriptRangeError :
    public SciEngErr {
public:
    SubscriptRangeError(int i, int n) : subscript(i), array_size(n) {}
    virtual String message() const {
        String msg;
        msg << "Subscript (" << subscript << ") out of range; ";
        msg << "must be between 0 and " << array_size - 1;
        return msg;
    }
private:
    const int subscript;
    const int array_size;
};

class ArraySizeError :
    public SciEngErr {
```

```
public:
    ArraySizeError(int n) : size(n) {}
    virtual String message() const {
        String msg;
        msg << "Array size " << size << " specified; must be non-negative";
        return msg;
    }
private:
    int size;
};
```

For brevity we show the functions inline here; under normal circumstances, these would be out of line.

Working with the interface allows us to ignore irrelevant differences among the exceptions. With these exception classes and functions, we can write a main() program for our GPIB simulator that traps simple errors:

ch9/tSimulator.C

```
try {
    MySimulators simulators;
    GPIBController_GIS gpib(simulators);
    Acme130_VS_GI_GC supply(gpib, 12);
    VoltyMetrics_VM_GI meter(gpib, 13);

    IVTester iv(supply, meter);
    double v_step = 1.0;
    for (int i = 0; i < 10; i++) {
        cout << iv.current(v_step * i) << endl;
    }

} catch(const SciEngErr& e) {
    cerr << e << endl;
}
```

Each error message can be created from information available where the error was detected. Moreover, multiple catch clauses can be used when we expect certain kinds of error and know how to recover from them.

9.11 Summary

This chapter was our first introduction to object-oriented programming. We started with a discussion of objects as representations and we discussed encapsulation of state and its relation to behavior. The bulk of the chapter described abstraction of common behavior to form interface base classes.

Just as designing and using physical objects involves thinking about idealizations or abstractions of the objects, designing and programming with C++ objects involves thinking about and using abstractions of C++ objects. C++ provides mechanisms beyond individual classes for expressing abstraction; we refer to them collectively as mechanisms for creating categories, groups of classes with something in common.

This chapter focused on one kind of commonality, common usability, and one kind of category, the interface category. An interface category is expressed in C++ as a group of classes having a common interface base class. This base class represents their common behavior as a list of member functions. Interface base classes are distinguished from other classes by the absence of member data and the declaration of their member functions with the keyword virtual. The action performed by such a virtual function is determined by the derived class, not by the base class.

We illustrated three ways that clients use interface base classes: Function clients call interface functions through references or pointers, class clients delay calls to interface functions by holding references or pointers, and collection clients delay and distribute calls to many interface functions by collecting references or pointers to many interfaces. We also demonstrated that objects may have more than one interface.

Interface categories are only one way of grouping classes, and the "is-usable-as" commonality among a group of classes is only one form of commonality. They work only when the common functions among a group of classes can be called with precisely uniform argument lists and return types. Virtual function interfaces are a powerful tool for a specific job. They do not work well as a means of expressing commonality of implementation (see Chapter 10) or commonality of structure (see Chapter 11).

9.12 Notes and Comments

9.1 The GPIB classes we have developed in this chapter are intended only to illustrate programming concepts and are not for practical use. Software packages for controlling GPIB-attached instruments are available commercially.

9.2 The style of C++ programming we used here to introduce virtual functions could be called programming with separation of interface and implementation. C++ does not require this style. Data members and virtual functions implementations can be mixed with declarations of virtual functions; client functions can be written for base classes with virtual functions that have implementations.

Some other languages allow (or require) their analogs of public member functions and private member data to be declared separately. For example, Modula-3 [57] has separately compiled interfaces and modules. This approach has the advantage that there is no need for a program that uses an object to be exposed to its representation.

This eliminates extraneous detail for the human reader, and, since the client program is compiled without access to the representation, client programs need not be recompiled when an object's representation is changed. However, this advantage must be balanced against the compiler's inability to exploit knowledge of object representations when compiling client code.

From the perspective of programming language design and implementation, choosing between the C++ approach and the Modula-3 approach has far-reaching consequences. The C++ approach admits more efficient implementations, at least with present compilation and linking techniques. Also see [44, Section 9.1c].

Many of the benefits of the Modula-3 approach can be had in C++ by programming with interface base classes. Interface base classes only specify interface; they have no data. A function, like checkCalibration(), that is written in terms of interface base classes need not be recompiled when object representations change. Avoiding recompilation can be a significant benefit when developing large programs.

9.3 Booch [16] offers an elaborate scheme for drawing objects and classes, as do Rumbaugh, *et al.* [97]. Although each has its merits for precise work, for sketches of objects we found them too complex. Our diagrams try to capture the main ideas of encapsulation and interface but are not suitable for the design of a complex system.

9.4 The relation among classes induced by derivation has been given various names in the object-oriented programming literature: *class hierarchy*, *inheritance hierarchy*, *class DAG*, *class lattice*, and probably others. The word *hierarchy* is inappropriate for use with C++ because it implies that each class has at most one base class; this is true in some other object-oriented languages, but not in C++. We like the phrase class lattice, which is used along with class DAG in [44], but a lattice is a mathematical structure with a specific meaning, a partially ordered set such that for any two elements there is a least upper bound and a greatest lower bound; classes and derivation do not have this property. Therefore we stick to class DAG.

9.5 Virtual function calls in a base-class constructor call the functions of the base class, not the functions of a partially constructed derived class object. A base class with pure virtual functions can call these functions in its constructor. C++ compilers can detect direct calls, but not calls to pure virtual functions from other functions called in a constructor. A call to a pure virtual function in this way will likely crash the program. Our separation of pure virtual functions into interface base classes and our guideline to avoid constructors in interface base classes prevent such illegal calls. ARM §

9.6 Pure virtual functions can be defined for base classes even though such definitions are not inherited. They may be used in derived classes by explicit qualification.

9.7 Our term category grew out of ideas in a remarkable computer algebra programming language called *Axiom* [60, 61]; we have mangled its concepts beyond recognition. We deliberately use category for four different and orthogonal kinds of metaclass commonality because we have observed that commonality arises first, and then programmers attempt to classify the commonality according to its appropriate implementation. Booch [16] uses the term *class category* to mean a logical collection of classes; this is not our use. The term *category* is also used in Smalltalk [49] with yet another meaning.

9.8 As a matter of design principle, we believe that interface base classes should be abstract base classes and vice versa. Technically and in practice, these ideas are not identical. Our interface base class is a design element, a base class with virtual functions and without implementation except in terms of the virtual interface. An abstract base class is a C++ language element, a base class with at least one pure virtual function. To be effective, interface base classes should be abstract; to be general and unconstrained, abstract base classes should be interface base classes. Other authors [16, 44, 114] use the term *abstract class* in the same way that we use the term interface class; we prefer not to confuse the design issue with the language component.

9.9 The creation/use dichotomy runs deep in C++: Given only a reference or pointer to an interface base class to use, one cannot, in general, create a new object as a copy of the one referred to. Solutions to this problem are discussed in Section 14.7.

9.10 The ANSI C++ standardization committee has recently relaxed the rule discussed on page 237 concerning the match between the declaration of a virtual function in a base class and an overriding declaration of that function in a derived class[98]. It is now permissible for the return types to differ if both return types are either pointers or references to classes and the class in the return type of the derived class virtual function is derived publicly from the class in the return type of the base class virtual function. Under the new rule, the following code, illegal under the old rule, would be allowed:

ch9/new-virt-ret.C

```
class C1 {};
class C2 : public C1 {};

class B {
public:
    virtual C1& f();
};

class D :
    public B {
public:
    virtual C2& f();    // Formerly illegal
};
```

9.11 The SimulatorFactory interface base is an example of a design pattern called an *abstract factory* [46]: It is an abstract interface for making certain kinds of objects. This use of abstraction allowed us to separate the choice of device simulators from the implementation of the GPIB controller simulator. We can change the choice of simulators without changing the controller simulator. This technique has been used on a much larger scale in the InterViews class library for creating user interfaces [72, 73], where the abstract factories are called *kits*.

9.13 Exercises

9.1 Supposing that the Acme 130 voltage supply operates from 0 to +25 volts, write the minimum() and maximum() member functions for Acme130. Compile and run code like that on page 228. Imagine that you have several dozen similar uses of the Acme130 class: How many will have to be changed when Acme sends an upgrade kit allowing your supply to work up to 50 volts?

9.2 Write class member functions for the version of VoltOn59 on page 232. Modify the version of checkCalibration() for Acme130 on page 232 to use this class.

9.3 Implement VoltOn59_VS_GI_GC, a version of VoltOn59 derived from VoltageSupply and GPIBInstrument that parallels the implementation of Acme130_VS_GI_GC on page 247.

9.4 Design an interface base class for voltmeters called Voltmeter that abstracts the part of VoltyMetrics (page 232) characteristic of all voltmeters. Write VoltyMetrics_VM_GI_GC, a version of VoltyMetrics, that derives from Voltmeter and from GPIBInstrument and parallels the implementation of Acme130_VS_GI_GC on page 247.

9.5 Sketch the VoltyMetrics_VM_GI_GC objects created in Exercise 9.4. Draw its class DAG.

9.6 Implement the member functions of the Acme130Simulation class on page 251. Implement ExperimentSimulation with test() returning the square of the voltage applied in apply(). Run the IVTester using this simulation.

9.7 Draw the class DAG for Acme130Simulation, VoltyMetricsSimulation, and ExperimentSimulation classes.

9.8 For simplicity, we lumped GPIB instruments that send data only, like voltmeters, with instruments that receive data only, like voltage supplies, into the GPIBInstrumentSimulation category. Redesign the GPIB simulator developed here to contain two arrays, one for GPIBTalker and GPIBListener objects.

9.9 Simulate a GPIB-connected thermometer such that its receive() returns the temperature in degrees K. Simulate an experiment in which the current-voltage curve of some device is measured as a function of temperature using one Acme 130 voltage supply for a heater control and one to apply the voltage for the IV curve. The ExperimentSimulation will need an applyHeat() and testTemperature() member function in this case.

9.10 Revise the try-block from page 109 to work with the exception interface in Section 9.10.

CHAPTER 10

Expressing Common Implementation

This chapter introduces C++ language features and programming techniques for expressing common data representations and function implementations to avoid replication of code. Replicating code is tedious, and replicated code is unnecessarily difficult to read: You have to read it all just to determine that it is replicated. Replication also increases the cost of change. Beyond having to change multiple copies of *similar* code, we *will*—sooner rather than later—fail to change one of these copies:

- Avoid replicated code.

Although this advice seems trivial, it works ultimately to make more code extensible and maintainable. Every time we recognize and capture commonality, we create code with fewer redundancies and more logical structure.

We begin our attack on replicated code with two simple techniques: extension by public inheritance, both without interfaces (Section 10.1) and with interfaces (Section 10.2). Extension by public inheritance is simple, powerful, and flexible but sometimes yields surprising behavior and can violate the information hiding principle by introducing unnecessary coupling among classes. We discuss these issues in Section 10.3. Section 10.4 introduces member function forwarding, factoring an implementation common to many classes into a single class and making that class a member of each of the original classes. Member function forwarding is more robust and couples program components less than public inheritance, but it has other drawbacks. Private inheritance, introduced in Section 10.5, provides most of the benefits of both approaches. Section 10.6 covers some of the details of inheritance, including scope, encapsulation, and their combination, access declarations. Finally, Section 10.7 links inheritance back to the interfaces from Chapter 9, and Section 10.8 lists the possible purposes for function declarations in a class.

10.1 Extension Using Public Inheritance

In Section 4.5 we wrote a class template called CheckedSimpleArray<T> that we said had the same data and some of the same functions as our earlier SimpleArray<T>. To write CheckedSimpleArray<T> we copied SimpleArray<T> and changed the name. Then we rewrote (page 108) one of the constructors, the projection operator member function, and the setSize() member function; we did not rewrite two other constructors, the destructor, the numElts() and copy() member functions, or the two assignment operator member functions. Future changes must be copied manually between these two classes.

In this section, we show how to express this kind of commonality without so much code replication. To simplify the example further, we discard the templatization and develop a CheckedFloatArray class as an extension of the SimpleFloatArray class on page 97; we'll extend the templatized version in Section 11.2.2.

10.1.1 Inheriting Member Functions

We extend SimpleFloatArray by deriving CheckedFloatArray from it:

ch10/CheckedFloatArray.h

```
class CheckedFloatArray :
    public SimpleFloatArray {
public:
    CheckedFloatArray(int n);                  // Create array of n elements
    CheckedFloatArray();                       // Create array of 0 elements
    float& operator[ ](int i);                 // Checked subscripting
    CheckedFloatArray& operator=(float);       // Scalar assignment

    class SubscriptRangeError {};              // Exception thrown for bad subscript
};
```

The public keyword that appears in the specification of the base class makes the base a *public base class* or simply a *public base,* and the derived class is said to be *derived publicly* from the base class. The SubscriptRangeError nested class defines the object that will be thrown when a subscript error is detected in the checked version; of course it would have some content in a real version.

Although we are using the same syntax and C++ mechanism for derivation that we introduced in Section 9.3.3, here the base class SimpleFloatArray is not an interface: It contains state, provides implementation, and does not declare virtual functions. To emphasize this distinction, we call it an *implementation base class,* or simply an *implementation base.*

An instance of the derived class contains an instance of its implementation base class, called a *base subobject,* as illustrated for our example in Fig. 10.1. Al-

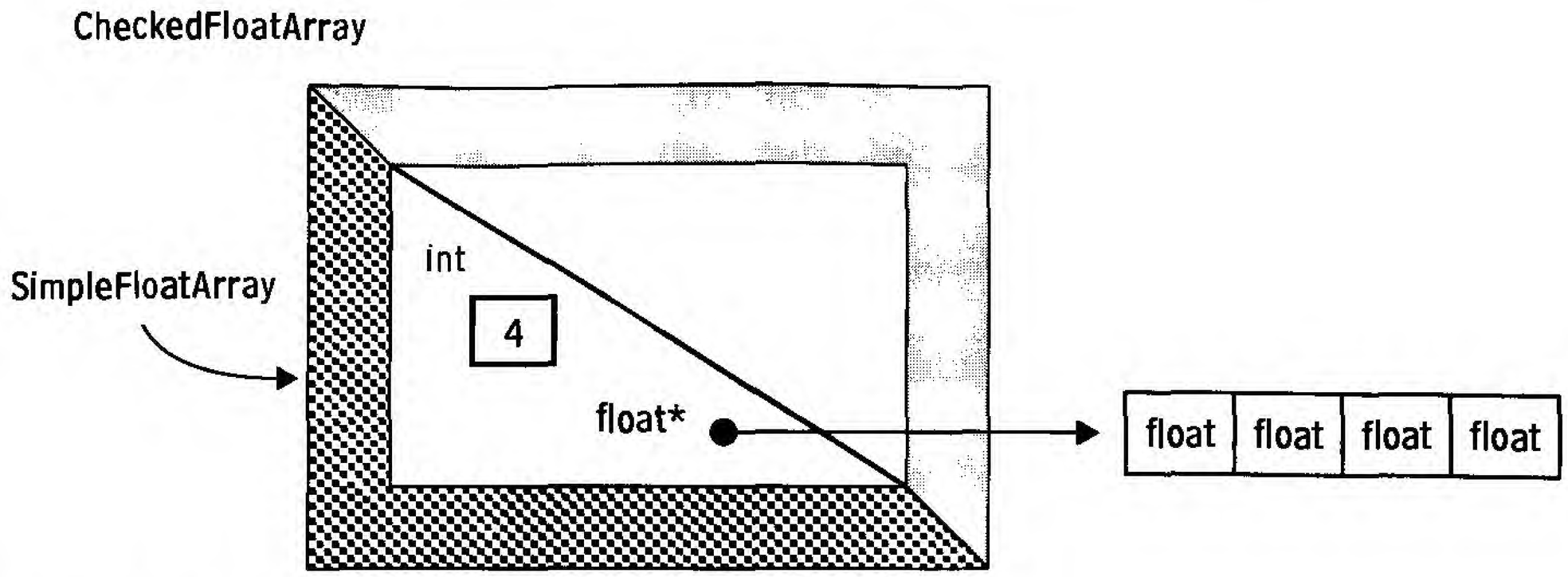


Figure 10.1 Object Sketch for CheckedFloatArray.

though there is one base subobject in this example, a derived class can have several base classes and hence several base subobjects.

Each member of a public base (other than constructors, destructors, and assignment operators) is inherited by the derived class, unless the derived class declares a member of the same name. Inherited member functions operate on their base subobject and can be used as if they were members of the base class. CheckedFloatArray derives publicly from SimpleFloatArray and therefore inherits its projection operator, numElts(), setSize(), and copy() member functions. These inherited functions are available without even being mentioned in the CheckedFloatArray class definition. ARM

We want the operations of SimpleFloatArray in CheckedFloatArray, but we want subscript access to be checked. To do this, we declared an operator[ ]() function in CheckedFloatArray. This function declaration *hides* the base class's operator[ ]() function and prevents it from being inherited. Instead CheckedFloatArray provides its own projection operator member function that provides the necessary checking:

ch10/CheckedFloatArray.h

```
float& CheckedFloatArray::operator[ ](int i) {
    if (i < 0 || i >= numElts()) throw SubscriptRangeError();
    return SimpleFloatArray::operator[ ](i);
}
```

The first statement throws an exception if the subscript is not valid. With a valid subscript, the second statement uses the scope resolution operator (::) to refer to the base class's projection operator function, calling it to access the appropriate array element. This technique provides a simple way for a derived class to add behavior to a base class.

10.1.2 Construction, Destruction, and Assignment

Constructors, destructors, and assignment operator functions (`operator=()`) are not inherited and receive special treatment. C++ will automatically define these functions for derived classes if we do not declare custom versions. For `CheckedFloatArray` we declared custom versions of constructors and assignment operators; we describe these first and then discuss the ones that C++ would have provided automatically.

Constructors initialize an object's state. The state of a derived class object includes any state the derived class itself introduces (none in this example) and the states of its base subobjects. Therefore a constructor in a derived class must invoke a constructor for each of its base subobjects and each of its members. Here is one of the `CheckedFloatArray` constructors we defined:

ch10/CheckedFloatArray.

```
CheckedFloatArray::CheckedFloatArray(int n) : SimpleFloatArray(n) {}
```

A base class constructor listed in this manner is called a *base initializer*. Base initializers are listed, along with any member initializers (cf. Section 6.2), in a comma-separated list following a colon (:) and preceding the opening brace of the constructor body. If a base or member has a default constructor, one that does not require arguments, the corresponding initializer may be omitted, as in this constructor:

¡12.6.2

ch10/CheckedFloatArray.h

```
CheckedFloatArray::CheckedFloatArray() {}
```

In such a case, the default constructor is called; for a built-in member, no initialization is done.

Consider the sequence of events following the allocation and preinitialization (Section 7.1) of an instance of `CheckedFloatArray`, as triggered by the statement:

ch10/tCheckedFloatArray.C

```
CheckedFloatArray a(10);
```

This statement calls the constructor for `CheckedFloatArray` that takes an `int` argument, here 10. This constructor specifies a base initializer for `SimpleFloatArray` passing an `int` argument, causing `SimpleFloatArray`'s constructor that takes an `int` (page 97) to be called. `SimpleFloatArray`'s constructor allocates the necessary built-in array of `float` elements and sets the two data members of the `SimpleFloatArray` base subobject. The important point is that the derived class's constructor invokes a base class's constructor to initialize the base subobject.

When the derived class has both bases and data members, the base subobjects are initialized before the data members are initialized; if there is more than one base, they are initialized in their order of appearance in the class definition. Be clear in your code:

¡12.6.2

- List base initializers in the order in which the bases appear in the class definition and before member initializers.

Otherwise you may be tempted to write expressions containing uninitialized data. In our experience, using inheritance to extend a class leads naturally to initialization of bases before data members.

After the base subobjects and data members have been initialized, the body of the constructor is run. The CheckedFloatArray constructors have empty bodies and thus have no effect. Exercise 10.1 gives an example with a nonempty constructor body.

If no constructor is declared, C++ generates a default constructor taking no arguments if it can. This function recursively calls base class and member object default constructors. If any subobject declares a private default constructor, C++ will not generate a default constructor for the derived class. ARM

In our example, we provided a default constructor because its automatic generation would have been suppressed by our constructor taking an integer argument. Default constructors are required to build arrays of class objects that are not initialized immediately: ARM

- Provide default constructors for classes with other constructors whenever possible.

If not declared in a class, C++ generates a copy constructor, a destructor, and a copy assignment operator (cf. Sections 6.3 and 6.6). The generated copy constructor and assignment operator recursively copy (assign) member objects and base subobjects. Since CheckedFloatArray declares neither a copy constructor nor a copy assignment operator, C++ generates them both. These generated functions call the corresponding base class functions to copy (assign) the base subobjects. Likewise C++ generates a destructor that invokes recursively the destructors for members and base subobjects. The C++-generated copy constructor, copy assignment operator, and destructor provide the behavior CheckedFloatArray needs, so there is no need to provide our own. ARM

However, since other constructors and assignment operators are neither inherited nor generated by C++, we declared and defined the other two constructors. Likewise we declared an assignment operator taking a float argument because assignment operator functions are not inherited. Here is its definition:

ch10/CheckedFloatArray.h

```
CheckedFloatArray& CheckedFloatArray::operator=(float rhs) {
    SimpleFloatArray::operator=(rhs);
    return *this;
}
```

Once again, we use the scope resolution operator to call a base class member to do the work.

10.1.3 Base Class Clients

An instance of a publicly derived class can be passed to a function that expects a reference (or pointer) to an instance of the base class, like this:

ch10/tCheckedFloatArray.C

```
float average(SimpleFloatArray& a) {
    double sum = 0.0;
    for (int i = 0; i < a.numElts(); i++) sum += a[i];
    return sum / a.numElts();
}

CheckedFloatArray a(10);
// ... set values in a
cout << average(a) << endl;
```

When average() is called with a CheckedFloatArray object, C++ arranges for it to be given a reference to the SimpleFloatArray base subobject. The code for average() is independent of whether the SimpleFloatArray it is working with is a base subobject; average() works with any SimpleFloatArray instance. In particular, when average() calls the projection operator, it calls SimpleFloatArray's projection operator, not CheckedFloatArray's projection operator: There is no subscript checking in average().

Public derivation in this example deliberately allows a CheckedFloatArray to be used as a SimpleFloatArray. We are allowing client functions to decide if they can ensure that subscript bounds will not be violated. Although this was a conscious design decision, the implementation of the decision only required listing SimpleFloatArray as a public base class of CheckedFloatArray. Thus in one statement we both inherit the implementation of SimpleFloatArray and allow the use of CheckedFloatArray via references to the SimpleFloatArray base subobject. We call this process *extension by public inheritance*.

In this section, we have applied extension by public inheritance to a concrete class, one not in an interface category. Despite its simplicity and power, extension by public inheritance has disadvantages, particularly for classes in an interface category. We discuss these disadvantages in Section 10.3 and discuss alternatives in the sections thereafter.

10.2 Extension of Interfaced Classes Using Public Inheritance

This section explores extension by public inheritance for interfaced classes, using the GPIB examples of Chapter 9 as the example. Suppose the Acme company

introduces an Acme 140 voltage supply identical to the Acme 130 except that the maximum possible output voltage is determined by the position of a jumper: In position "J1" the maximum output is 10 volts; in position "J2" the maximum output is 50 volts. How do we write an Acme140 class that represents an Acme 140 voltage supply?

We could copy the latest version of our Acme130 class (Acme130_VS_GI_GC), change all occurrences of "130" to "140," add a data member to hold the jumper setting, and rewrite the maximum() member function to return the appropriate value depending on the jumper setting. Although expedient, this replicates code and loses any connection between Acme130 and Acme140, a connection that we may (or may not) want.

A better alternative is to proceed as we did in Section 10.1 and derive Acme140 from Acme130_VS_GI_GC:

ch10/Acme140.h

```
class Acme140 :
    public Acme130_VS_GI_GC {
public:
    enum Jumper { J1, J2 };   // Possible jumper settings

    Acme140(GPIBController& controller, int gpib_address, Jumper j);
    virtual float maximum() const;
private:
    float max_voltage;
};
```

The resulting object is illustrated for our example in Fig. 10.2. Compare this sketch with Fig. 9.6. An Acme140 is usable as an Acme130_VS_GI_GC in much the same way that an Acme130_VS_GI_GC is usable as a VoltageSupply or a GPIBInstrument.

The "is-usable-as" relation is transitive: If a C is usable as a B, and a B is usable as an A, then a C is usable as an A. In terms of a class DAG, a C is usable as an A if there is a path (attending to the direction of the arrows) in the DAG from C to A. Thus an Acme140 is usable as an Acme130_VS_GI_GC, a VoltageSupply, and a GPIBInstrument, as illustrated by the class DAG shown in Fig. 10.3. Figure 10.2 also illustrates that an Acme 140 is usable as both a VoltageSupply and as a GPIBInstrument by showing both interfaces on the outside of the Acme140 object.

Although an Acme140 is usable as an Acme130_VS_GI_GC, the Acme130_VS_GI_GC class is an implementation base, not an interface, because it contains state and provides implementation. Thus, as in the preceding section, we must provide a constructor to initialize the implementation's state:

ch10/Acme140.h

```
Acme140::Acme140(GPIBController& controller, int gpib_address, Jumper j) :
    Acme130_VS_GI_GC(controller, gpib_address),
    max_voltage( j == J1 ? 10.0 : 50.0) {
}
```

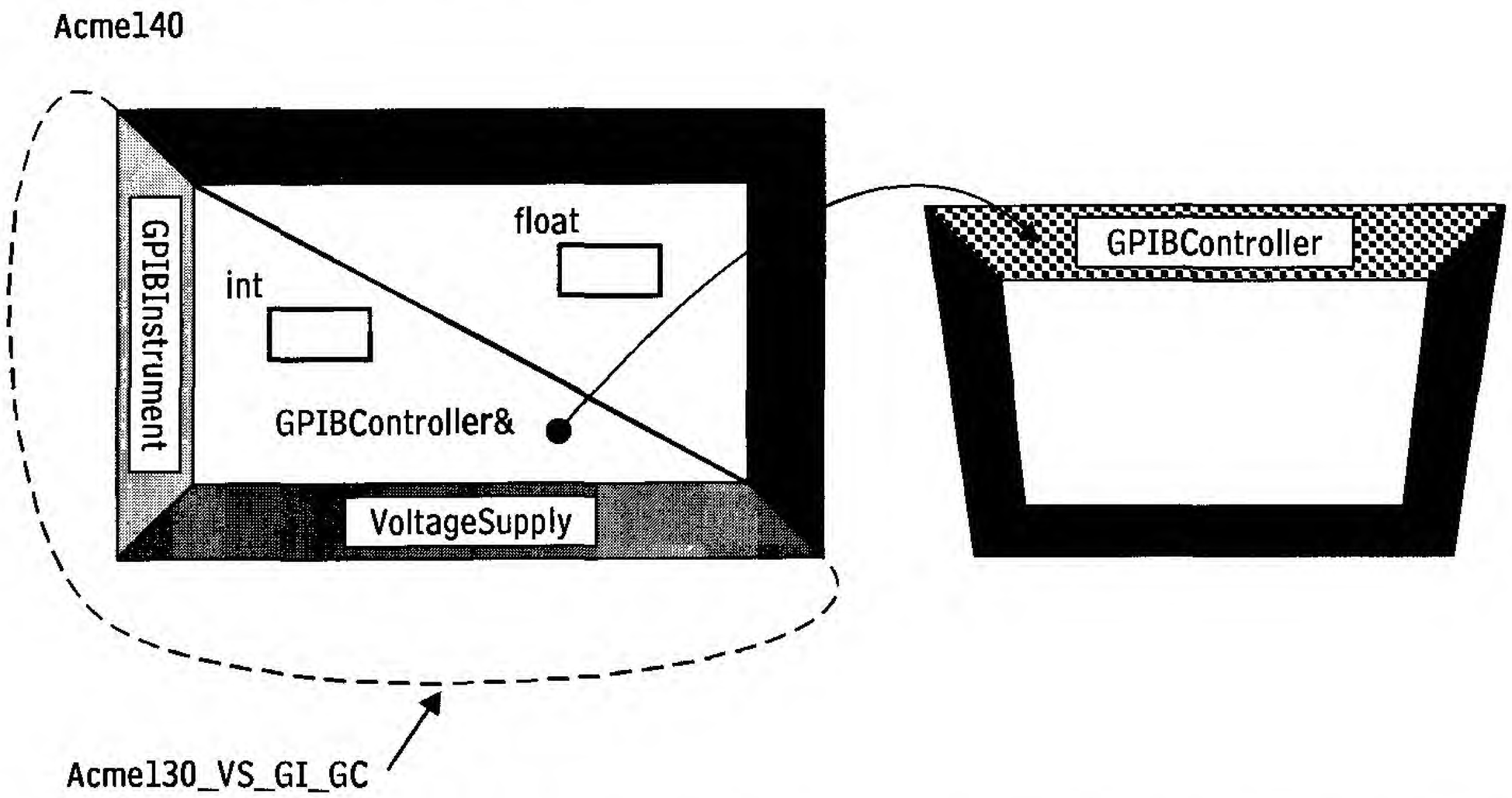


Figure 10.2 Object Sketch for Acme140. The contained objects are the Acme130_VS_GI_GC base subobject and the max_voltage data member.

This constructor initializes both the base subobject and the member datum max_voltage.

Member functions of a public base (other than constructors, destructors, and assignment operators) are inherited by the derived class. Acme140 derives publicly from Acme130_VS_GI_GC and therefore inherits its member functions. Thus we need not replicate any code to make the Acme140 behave like an Acme130; to make it behave like an Acme140 we have to change the behavior of the maximum() function like this:

ch10/Acme140.

```
float Acme140::maximum() const {
    return max_voltage;
}
```

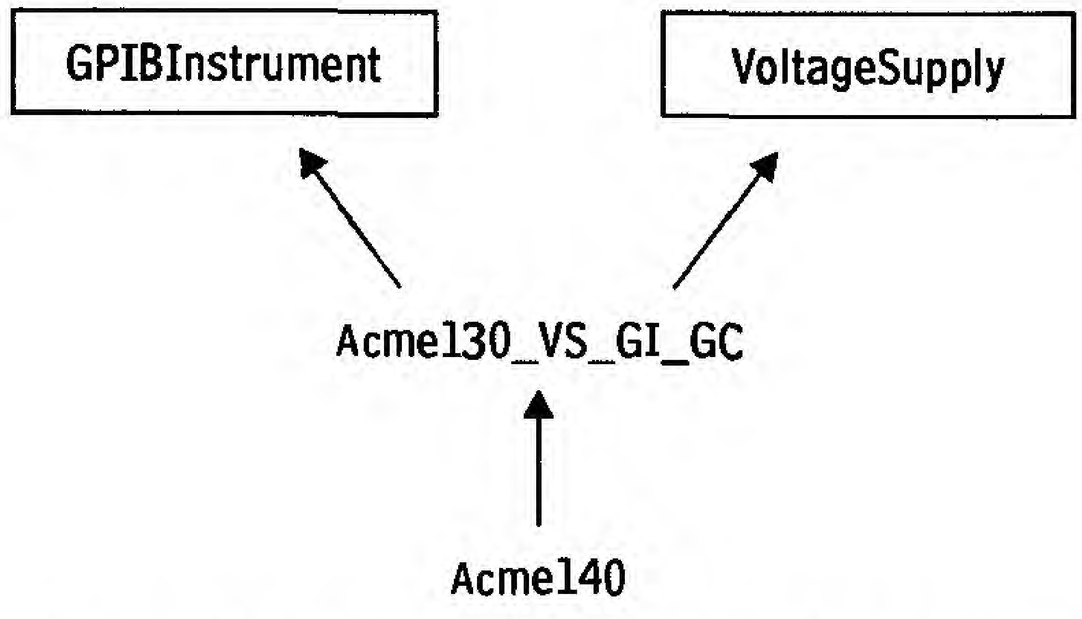


Figure 10.3 Class DAG for Acme140. Classes enclosed in rectangles are interface base classes.

When a derived class declares a virtual member function with the same name and argument types as a virtual function in the base class, the derived class's function is said to *override* the base class's function.

Calling a virtual function for an instance of a derived class invokes the overriding derived-class function definition. Here the Acme140 member function maximum() overrides the Acme130_VS_GI_GC member function maximum(). Let's look at some examples:

ch10/tAcme140.C

```
GPIBController_GC gpib;
Acme140 supply1(gpib, 12, Acme140::J2);
Acme140 supply2(gpib, 13, Acme140::J1);
Acme130_VS_GI_GC supply3(gpib, 14);

// Use directly
cout << supply1.maximum() << endl;          // Displays 50
cout << supply2.maximum() << endl;          // Displays 10
cout << supply3.maximum() << endl;          // Displays 10

// Use as Acme130_VS_GI_GC
Acme130_VS_GI_GC& s1_as_130 = supply1;
Acme130_VS_GI_GC& s2_as_130 = supply2;
Acme130_VS_GI_GC& s3_as_130 = supply3;
cout << s1_as_130.maximum() << endl;        // Displays 50
cout << s2_as_130.maximum() << endl;        // Displays 10
cout << s3_as_130.maximum() << endl;        // Displays 10

// Use as VoltageSupply
VoltageSupply& v1 = supply1;
VoltageSupply& v2 = supply2;
VoltageSupply& v3 = supply3;
cout << v1.maximum() << endl;               // Displays 50
cout << v2.maximum() << endl;               // Displays 10
cout << v3.maximum() << endl;               // Displays 10
```

In each case the answers were the same and depended on the actual object, not on the interface being used. When a virtual function (like maximum()) is called, the function invoked depends on the type of the object, not on the type of reference for which the function is called. (Contrast this with the behavior described in Section 10.1.3.)

Importantly, this behavior obtains even when a function supplied by the base class calls a function that is overridden by the derived class. For example, the set() function in Acme130_VS_GI_GC calls maximum():

ch10/Acme130_VS_GI_GC.C

```
void Acme130_VS_GI_GC::set(float voltage){
    if (voltage > maximum() || voltage < minimum()) throw "Acme 130 voltage out of range";
    send(voltage);
}
```

Acme140 does not override set(): This is the function that will be called for an Acme140 object. However, Acme140's maximum() is called when set() is called for an Acme140 object, even though Acme130_VS_GI_GC's set() is used. Continuing the previous example, in which supply1 provided a 50-volt maximum and supply2 provided a 10-volt maximum, we see

ch10/tAcme140.C

```
supply1.set(15);        // Works
supply2.set(15);        // Throws an exception
s1_as_130.set(15);      // Works
s2_as_130.set(15);      // Throws an exception
v1.set(15);             // Works
v2.set(15);             // Throws an exception
```

This behavior is "object oriented": The maximum() function called in set() depends on the object with which set() is called. This behavior is a consequence of declaring maximum() to be a virtual function.

10.3 Problems with Public Inheritance

As we have seen in Sections 10.1 and 10.2, public inheritance is an effective way to avoid replicating code. We were able to write a new class that was almost the same as an existing class by extending and modifying the behavior of the existing class. Let's now look at this process from several viewpoints.

First let's look at what we've done in terms of information hiding. Because Acme140 is derived publicly from Acme130_VS_GI_GC, an Acme140 object is usable as an Acme130_VS_GI_GC object. This exposes to all clients our decision to implement Acme140 by extending Acme130_VS_GI_GC. Any code that exploits the fact that an Acme140 is usable as an Acme130_VS_GI_GC would have to be changed if we changed our implementation decision.

- Public derivation from an implementation base exposes an implementation decision.

Whether this is acceptable depends on the likelihood that the implementation decision will change. Remember, the information hiding principle (Section 8.3) says to hide decisions that are likely to change, not to hide all decisions. In practice, you must balance the ease of extension by public inheritance against the product of the cost of changing the implementation and the likelihood of change.

As we will see later in this chapter, with a bit more work it is possible to keep the implementation decision a secret, making the cost of change small.

Now let's look at what we've done from the viewpoint of the requirements imposed on the base class. Had Acme130_VS_GI_GC's maximum() function not been virtual, we would have been unable to use extension by public inheritance. With non-virtual functions that are declared in both the base and derived classes, the actual function invoked is determined by the type of the reference or pointer used in the function call, not by the object's type. So if maximum() had not been virtual, calling it when using an Acme140 object as an Acme130_VS_GI_GC (e.g., s1_as_130) would yield 10 instead of the (correct) 50 obtained when using the object as an Acme140 (e.g., supply1).

Meyers [82, p. 131] calls this behavior of non-virtual functions "schizophrenic": An object responds differently to what appears to be the same function call depending on the type of the pointer to the object used in the call. This behavior is not "object oriented." As a source of surprising results, we like to avoid this behavior:

- Avoid new definitions in derived classes for functions declared non-virtual in a public base class.

Turned around, this guideline means that public base classes impose limitations on the function names we can use in derived classes: Non-virtual member functions limit extension by public inheritance. Thus we conclude the following:

- When using public inheritance, declare base class member functions virtual unless there is a reason not to.

There are two reasons not to declare a member function virtual. One has to do with the intent of the class, the other with the intent of the function itself. If you are writing a *concrete class*, a class intended to provide a single capability with maximum efficiency and not to fit into a larger system of classes, it is likely that all of the member functions should be non-virtual to avoid the overhead of virtual function calls (see Notes and Comments 10.1).

Even when writing a class designed to be part of a larger system of classes, there are occasions when you want to constrain derived classes to use the base class behavior for a particular member function. For example, the reset() function described on page 242 effectively expresses the constraint that all VoltageSupply objects respond in fixed fashion to the reset() function call: reset means "set yourself to your minimum voltage." By declaring reset() non-virtual, its definition will be fixed for all VoltageSupply objects called through the VoltageSupply interface.

Finally, let's look at the Acme140 example from the viewpoint of its clients. We have already seen that an Acme140 object is usable as an Acme130_VS_GI_GC. Precisely, this means that a reference (pointer) to an Acme130_VS_GI_GC can refer

(point) to an Acme140 object and that all of the member functions specified by Acme130_VS_GI_GC can be invoked through such a reference or pointer. So far, so good. Now let's write a function to display the maximum output voltage for an Acme130_VS_GI_GC object:

ch10/tAcme140.C

```
void display_max_voltage(Acme130_VS_GI_GC s) {
    cout << "Maximum output voltage is " << s.maximum() << endl;
}
```

Now we continue the example on page 273:

ch10/tAcme140.C

```
display_max_voltage(supply1);    // Displays: Maximum output voltage is 10
display_max_voltage(supply2);    // Displays: Maximum output voltage is 10
```

Something is wrong: supply1 has a maximum output voltage of 50.0, not 10.0 as displayed.

The origin of this result is the argument of display_max_voltage(): It is an Acme130_VS_GI_GC, not a reference to one. When we pass an Acme140 object, Acme130_VS_GI_GC's (C++-generated) copy constructor is called to make a copy of the argument (cf. Section 6.3). The first argument of a copy constructor is always a reference, so this copy constructor's argument is a reference to an Acme130_VS_GI_GC; an Acme140 is usable as an Acme130_VS_GI_GC, so the copy constructor is run, creating an Acme130_VS_GI_GC object. This is not an Acme140 object, it has no member to hold a jumper setting, and it has the base class definition of maximum(). The maximum output voltage for this object is displayed.

Our mistake was to forget that the "is-usable-as" relation is only effective for objects used through references and pointers. Since Acme130_VS_GI_GC supplies both interface and implementation, the compiler cannot catch our erroneous attempt to exploit an "is-usable-as" relation directly with objects instead of through references or pointers.

Contrast this with the behavior of abstract base classes: Had we tried to write display_max_voltage() with a VoltageSupply argument, the compiler would have caught our error because no instances of an abstract base can be created. The moral is this:

- Separate interface and implementation; use abstract base classes to specify interface.

This advice need not and cannot always be followed. When the design of a class dictates the combination of inheritance from a base class and being usable through that base class, as in the CheckedFloatArray example at the beginning of this chapter, public inheritance is ideal. Most such examples occur when we must have maximum performance at the expense of maintainability or adaptability. When expedient implementation of a class requires derivation from a base class that

combines interface and implementation, no separation is possible. In other cases, consider alternatives like the techniques described in next two sections.

10.4 Member Function Forwarding

Consider the classes from Chapter 9 that represent instruments connected via the GPIB: Acme130_VS_GI_GC (page 247), VoltOn59_VS_GI_GC (Exercise 9.3), and VoltyMetrics_VM_GI_GC (Exercise 9.4). All three classes have both an int member that holds the GPIB address and a reference to an object in the GPIBController interface category. Each class uses these members in an identical way to implement the functions specified by the GPIBInstrument interface. Here, for example, are their implementations of send():

ch10/Acme130_VS_GI.C

```
void Acme130_VS_GI::send(const char* cmd) {
    my_controller.send(my_gpib_address, cmd);
}
```

ch10/VoltOn59_VS_GI.C

```
void VoltOn59_VS_GI::send(const char* cmd) {
    my_controller.send(my_gpib_address, cmd);
}
```

ch10/VoltyMetrics_VM_GI.C

```
void VoltyMetrics_VM_GI::send(const char* cmd){
    my_controller.send(my_gpib_address, cmd);
}
```

To avoid this replication of both data specification and code, we group the address and the controller reference as member data in a new class and implement the GPIBInstrument functions in that class.

We call our new class GPIBInstrumentData, indicating that it implements the GPIBInstrument interface. Here is one version:

ch10/GPIBInstrumentData.h

```
class GPIBInstrumentData {
public:
    GPIBInstrumentData(GPIBController& c, int address, const char* name);

// GPIBInstrument interface
    void send(const char*);      // Command.
    void send(float f);          // Command with value.
    float receive();             // Data point.

private:
    GPIBController& my_controller;
    int my_gpib_address;
};
```

Note that this class is structurally similar to the Acme130_VS_GI_GC, VoltOn59_VS_GI_GC, and VoltyMetrics_VM_GI_GC classes and not at all like the VoltageSupply or GPIBInstrument interface base classes. GPIBInstrumentData does not specify; instead, it implements.

The implementations of the functions specified by GPIBInstrument are essentially the same as those we saw for Acme130_VS_GI (page 244):

ch10/GPIBInstrumentData.C

```
void GPIBInstrumentData::send(float value) {
    my_controller.send(my_gpib_address, value);
}

void GPIBInstrumentData::send(const char* cmd) {
    my_controller.send(my_gpib_address, cmd);
}

float GPIBInstrumentData::receive() {
    return my_controller.receive(my_gpib_address);
}
```

However, a single version of these functions will now work for all the GPIB instrument objects.

The GPIBInstrumentData constructor initializes the member data and inserts the device into the controller, like the Acme130_VS_GI_GC constructor (page 248):

ch10/GPIBInstrumentData.C

```
GPIBInstrumentData::
GPIBInstrumentData(GPIBController& c, int address, const char* name) :
    my_controller(c),
    my_gpib_address(address) {
    my_controller.insert(name, my_gpib_address);
}
```

The only difference is that the constructor, not belonging to a specific device, doesn't know the device name; instead the device name is passed as an argument.

Having recognized and extracted the common implementation aspects of the classes in the GPIBInstrument interface, we can recode the original classes to exploit the commonality. We'll start with Acme130_VS_GI_GC and will call the revised version Acme130_Fwd, where the _Fwd means "version with forwarding":

ch10/Acme130_Fwd.h

```
class Acme130_Fwd :
    public VoltageSupply,
    public GPIBInstrument {
```

```
public:
    Acme130_Fwd(GPIBController& controller, int gpib_address) :
        gpib_rep(controller, gpib_address, "Acme130_Fwd") {}

// VoltageSupply interface
    virtual void set(float volts);
    virtual float minimum() const;
    virtual float maximum() const;

// GPIBInstrument interface
    virtual void send(const char* cmd) { gpib_rep.send(cmd);       }
    virtual void send(float f)         { gpib_rep.send(f);         }
    virtual float receive()            { return gpib_rep.receive(); }

private:
    GPIBInstrumentData gpib_rep;
};
```

Each GPIBInstrument function is implemented by calling the corresponding member function for the encapsulated gpib_rep object. The Acme130_Fwd constructor initializes the controller and address indirectly, by initializing the gpib_rep object, instead of directly, as in the Acme130_VS_GI_GC constructor.

Each function specified by the GPIBInstrument is implemented by a *forwarding function*, a function that forwards the call to a data member, here gpib_rep. This technique is called *member function forwarding* or *delegation*. We wrote the member functions directly in the class body to emphasize their trivial nature, but it is generally better to provide out-of-line definitions; see Notes and Comments 10.2.

Figure 10.4 illustrates this version of the class. The object's interface is identical to the interface in Fig. 9.4. Only the implementation has changed. No code written with references to GPIBInstrument or VoltageSupply objects need be revised or even recompiled. When we complete the revision of VoltOn59 and VoltyMetrics classes (see Exercise 10.4), we will have achieved our goal of avoiding replication of data specification and function implementation.

Member function forwarding gets the job done: The member data are encapsulated and virtual function interfaces are provided and implemented without function duplication beyond the trivial forwarding functions. However, member function forwarding suffers from two practical problems.

First, the forwarding functions, however trivial, must be written. This means they must be maintained as identical text in multiple classes, the very problem we sought to avoid, albeit now on a smaller scale.

Second, if a class derived from Acme130_Fwd overrides one of the forwarding functions (which are virtual), the behavior of the other forwarded functions will

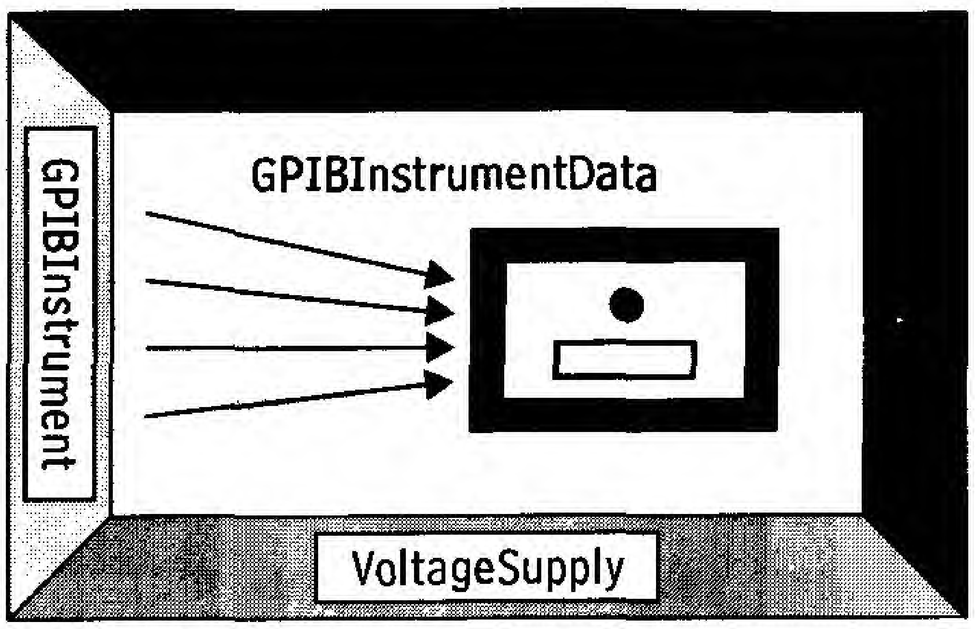


Figure 10.4 An Object with Two Interface Base Classes. The GPIBInstrument interface is implemented with member function forwarding to the internal GPIBInstrumentData object.

not be affected. In the present example, the three forwarded functions do not call each other. But consider the set() function on page 274. This function relies on the values returned by maximum() and minimum(). If we wrote a class that forwarded set() to an Acme130_VS_GI_GC object, the minimum and maximum voltage would always be those of Acme130_VS_GI_GC, not of the new class. See Exercise 10.6.

The technique we describe in the next section avoids these problems. In Section 13.6.1 we discuss an important case in which forwarding is the only available solution.

10.5 Private Inheritance for Implementation

To capture the commonality expressed in GPIBInstrumentData without the problems of member function forwarding, we use derivation. Rather than having a GPIBInstrumentData data member in each instrument class, we derive each instrument class from (a modified version of) the GPIBInstrumentData class. Our goal in this section is to describe a C++ idiom that says "use GPIBInstrumentData to implement the interface specified by GPIBInstrument."

The first step is to modify GPIBInstrumentData to connect it to GPIBInstrument, to say that a GPIBInstrumentData is usable as a GPIBInstrument. We'll call this version GPIBInstrumentData_GI, to emphasize the connection. As usual, the "is-usable-as" relation is specified by deriving publicly:

ch10/GPIBInstrumentData_GI.h

```
class GPIBInstrumentData_GI:
    public virtual GPIBInstrument {
public:
    GPIBInstrumentData_GI(GPIBController& c, int address, const char* name);
```

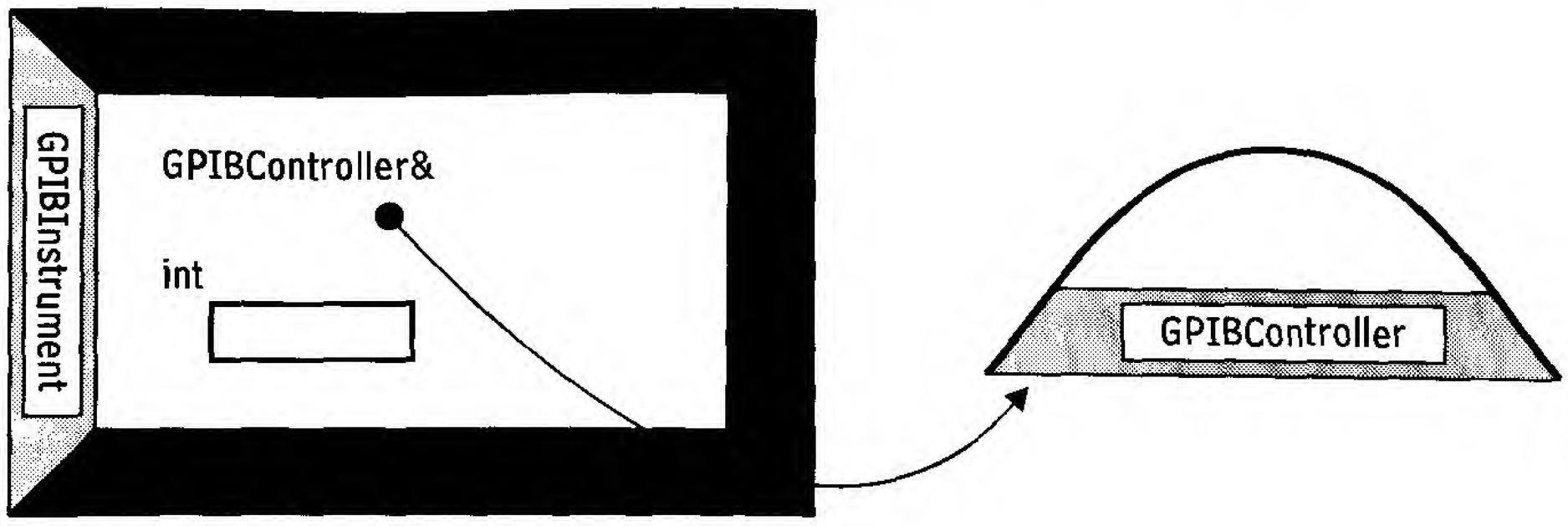


Figure 10.5 Sketch of a GPIBInstrumentData_GI Object.

```
// GPIBInstrument interface
     virtual void send(const char*);      // Command.
     virtual void send(float f);          // Command with value.
     virtual float receive();             // Data point.
private:
     GPIBController& my_controller;
     int my_gpib_address;
};
```

We have not encountered previously the virtual keyword used in the list of base classes, as in public virtual GPIBInstrument. Its meaning is explained later. When virtual is specified, we say that the derived class is *derived virtually* from the base class and the base class is called a *virtual base class*, or simply a *virtual base*.

A GPIBInstrumentData_GI object is sketched in Fig. 10.5. In contrast to the sketch in Fig. 10.4 of the GPIBInstrumentData member datum inside an Acme130_Fwd, this GPIBInstrumentData_GI object has a GPIBInstrument interface. The implementations of GPIBInstrumentData_GI's members functions (not shown) are identical to the implementations of the corresponding functions of our original GPIBInstrumentData on page 277.

The second step is to modify a version of Acme130 to implement its GPIBInstrument interface with GPIBInstrumentData_GI. We modify Acme130_VS_GI_GC (page 247) to obtain Acme130_GIData, where the GIData means "version with GPIBInstrumentData":

ch10/Acme130_GIData.h

```
class Acme130_GIData :
     public virtual VoltageSupply,
     public virtual GPIBInstrument,
     private GPIBInstrumentData_GI {
```

```
public:
    Acme130_GIData(GPIBController& controller, int gpib_address);

// VoltageSupply interface
    virtual void set(float volts);
    virtual float minimum() const;
    virtual float maximum() const;
};
```

Again, we have derived virtually from the interface bases; we have also derived privately from GPIBInstrumentData_GI, as indicated by the use of the keyword private in the base class specification. A base class derived from privately is called a *private base class* or simply a *private base*.

As with public derivation, an instance of a class derived from a private base contains a base subobject (cf. Section 10.1). Unlike public derivation,

- Private derivation does not establish an "is-usable-as" relation.

In our example, an Acme130_GIData is usable as a VoltageSupply and as a GPIBInstrument, but not as a GPIBInstrumentData_GI. This is what we want:

- Private derivation establishes an "is-implemented-in-terms-of" relation but keeps that relation secret.

Using private derivation hides our decision to implement the GPIBInstrument interface using a GPIBInstrumentData_GI base subobject.

10.5.1 How Virtual Base Classes Work

The powerful combination of public derivation from interfaces and private inheritance of implementation is unlike anything you are likely to have experience with from procedural programming. Here we take a closer look at this technique.

Notice that Acme130_GIData does not declare the member functions specified by the GPIBInstrument interface; they are inherited from GPIBInstrumentData_GI. However, GPIBInstrumentData_GI is a private base and client functions are blocked from accessing directly these inherited functions. The conduit for outside access is made when Acme130_GIData derives publicly and virtually from the same GPIBInstrument interface that GPIBInstrumentData_GI uses. The resulting object and class DAG is illustrated in Fig. 10.6. Notice that the GPIBInstrument interface appears only one place in each diagram. The virtual base classes are shared by all classes in the class graph: GPIBInstrumentData_GI shares its GPIBInstrument interface with Acme130_GIData. When function calls come in through this public interface, C++ routes the calls to the encapsulated GPIBInstrumentData_GI subobject.

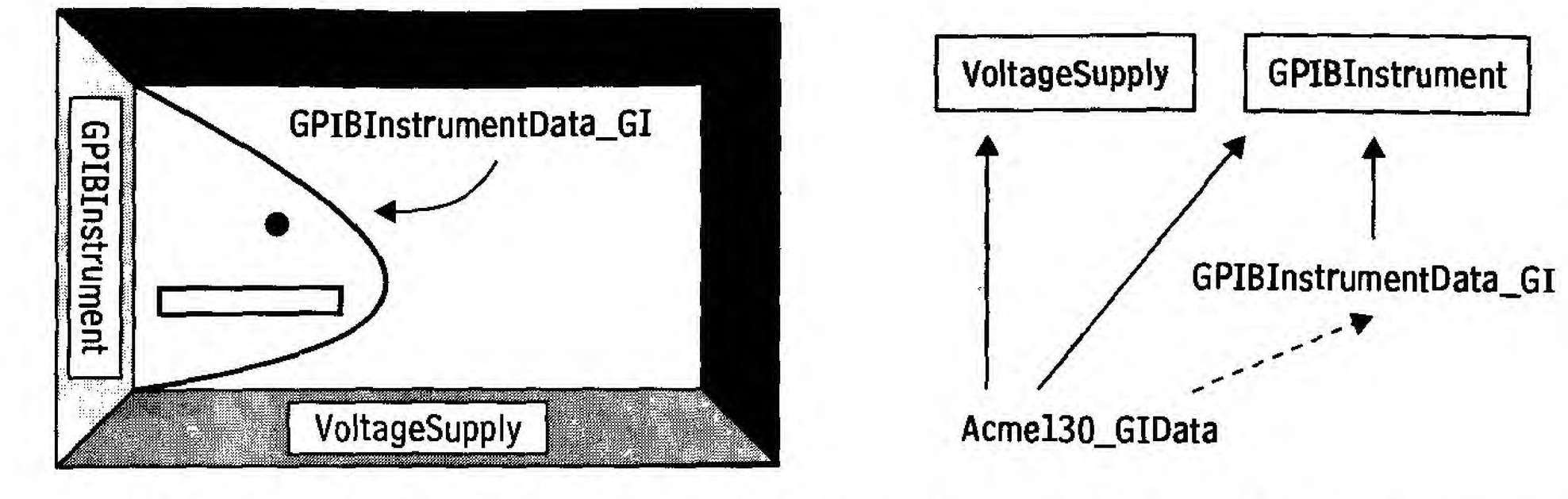


Figure 10.6 Object Sketch and Class DAG for Class Acme130_GIData. The dashed arrow indicates private derivation.

To understand virtual derivation better, let's look at the effect of omitting either or both of the virtual keywords in the derivations from GPIBInstrument. Doing so would result in an object with two independent GPIBInstrument interfaces, as shown in Fig. 10.7. With such a structure, a client calling a member function in the GPIBInstrument interface is ambiguous: Which of the two GPIBInstrument interfaces is desired? It is not illegal to create such a structure, but the compiler will flag as an error any ambiguous call to a member function.

Let's return to the correct structure, the one with virtual derivation. Member function calls through the GPIBInstrument interface go directly to the member functions provided by GPIBInstrumentData_GI. This is arranged automatically by C++; no forwarding functions are necessary. The functions inherited from

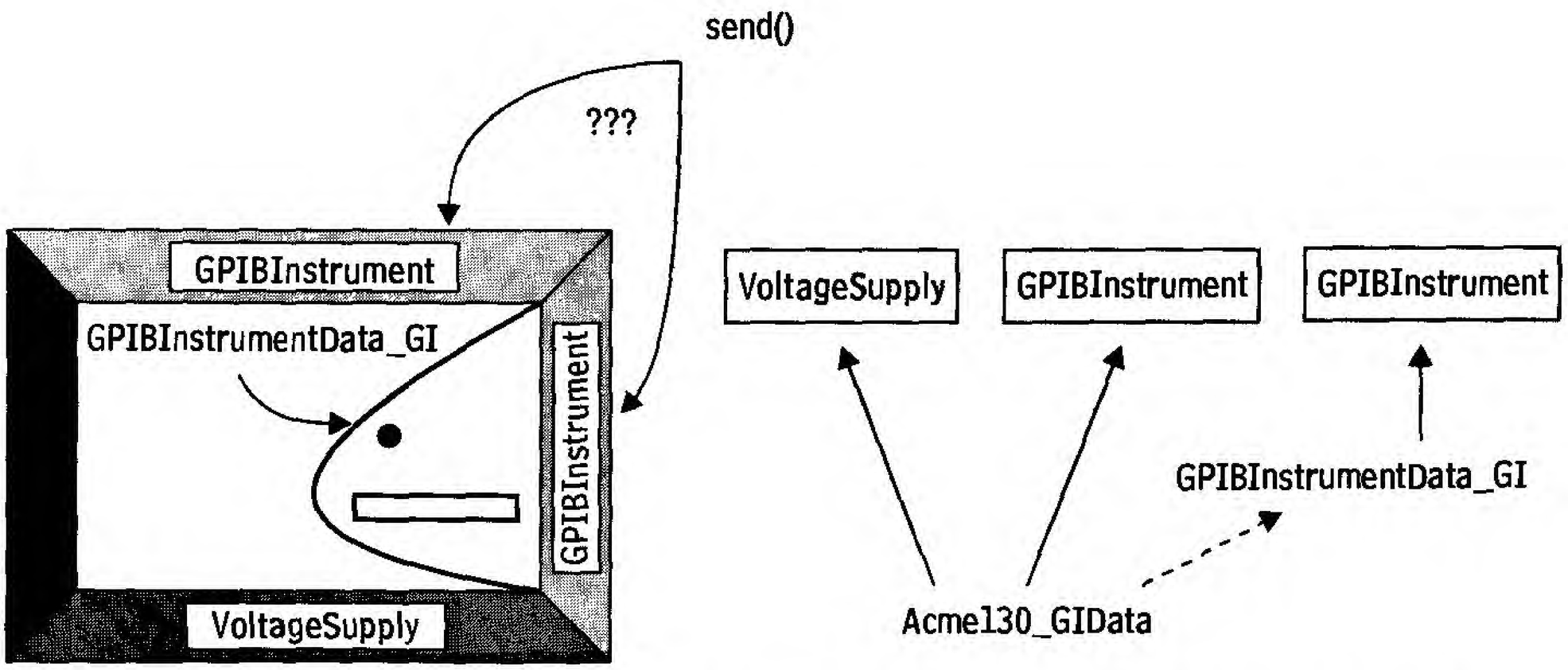


Figure 10.7 Object Sketch and Class DAG That Would Result If Either Derivation from GPIBInstrument in the Definition of Acme130_GIData Were Non-virtual.

GPIBInstrumentData_GI implement the specification of the single GPIBInstrument interface. Thus the procedure for implementing an interface using inheritance uses virtual base classes for the interfaces and multiple derivation to combine the interface and implementation.

In other respects, the Acme130_GIData class is similar to the other classes we have discussed. The Acme130_GIData constructor takes the same argument list as earlier versions of the Acme 130 class. These arguments, with the name of the instrument, are passed on to initialize the GPIBInstrumentData_GI class:

ch10/Acme130_GIData.C

```
Acme130_GIData::Acme130_GIData(GPIBController& controller, int gpib_address) :
    GPIBInstrumentData_GI(controller, gpib_address, "Acme130_GIData") {
}
```

GPIBInstrument need not be initialized because it is an interface and has no state.

Using private inheritance for implementation, we have achieved both encapsulation and reuse of implementation. Using private derivation keeps our choice of implementation secret: An Acme130_GIData is not usable as a GPIBInstrumentData_GI, and the compiler will reject any attempt to write client functions using references to GPIBInstrumentData_GI. No forwarding functions need be written or maintained, and the problems we encountered in Section 10.4 due to the interaction of forwarding functions and overriding are eliminated.

- Implement interfaces using private inheritance with virtual derivation from interface bases.

If, in Acme130_GIData, we want to override one of the member functions provided by GPIBInstrumentData_GI, we can do so by declaring the function to be overridden and supplying a definition; see Exercise 10.7.

To say simply that Acme130_GIData "inherits" the members of GPIBInstrumentData_GI leaves many details to explore. In the next section, we study the important details. The details do not alter the basic idea: A derived-class specification contains member data specified in all of its base classes and the member functions of a derived class include the member functions of all of its base classes. The details allow us to control the inheritance process, to merge selectively members from base classes into a derived class.

10.6 Mechanics of Inheritance

We have explored in this and the preceding chapter various uses of derivation and inheritance: interfaces, extension, and implementation of interfaces. All of these uses are supported by a unified mechanism described in this section.

Many of the mechanics of inheritance in C++ work with *names*. We emphasize the word name: The mechanics of inheritance do not work on functions or vari-

ables or classes, but on names. A name is an identifier—the usual string of characters without spaces—with a meaning. As the compiler analyzes classes and functions you have defined, it looks up or *resolves* identifiers it encounters to match the names it has seen previously. The resolution of an identifier determines its meaning.

Each name is introduced by a declaration. The nesting of the declaration in classes, functions, or other blocks determines the name's scope (see Section 7.3.1); the position of the declaration in a class DAG determines whether this name will dominate or be dominated by other names with the same identifier; and access specifications (e.g., public, private) preceding the declaration determine what parts of the program will be able to access this name. Identifiers must resolve to a name in the scope being compiled, the name must be one kind of C++ entity unambiguously, and the section of code being compiled must have access to the name it matches. These topics—scope, dominance, and access—are covered in the following subsections.

To explore these topics we use a slightly more elaborate example. Imagine that you obtain the Acme company's latest voltage supply, the Acme 230. This supply is plug compatible with the Acme 130 supply, but it has an additional front-panel switch for a 10× range and two more GPIB commands, "RNG0" for 0-10 volts and "RNG1" for 0-100 volts. On low range it operates on the same 0-10-V range of the Acme 130, but on high range it operates from 0-100 V with some loss in precision.

An object to represent a model 230 might build on Acme130_GIData in the same fashion that Acme130_GIData built on GPIBInstrumentData_GI, adding features for the dual range:

ch10/Acme230.h

```
class Acme230 :
    public virtual VoltageSupply,
    public virtual GPIBInstrument,
    private Acme130_GIData {
public:
    Acme230(GPIBController& which_GPIB, int what_address);

    virtual void set(float volts);
    virtual float maximum() const;

    enum Range { high, low };
    virtual void set(Range r);
private:
    Range range_setting;
};
```

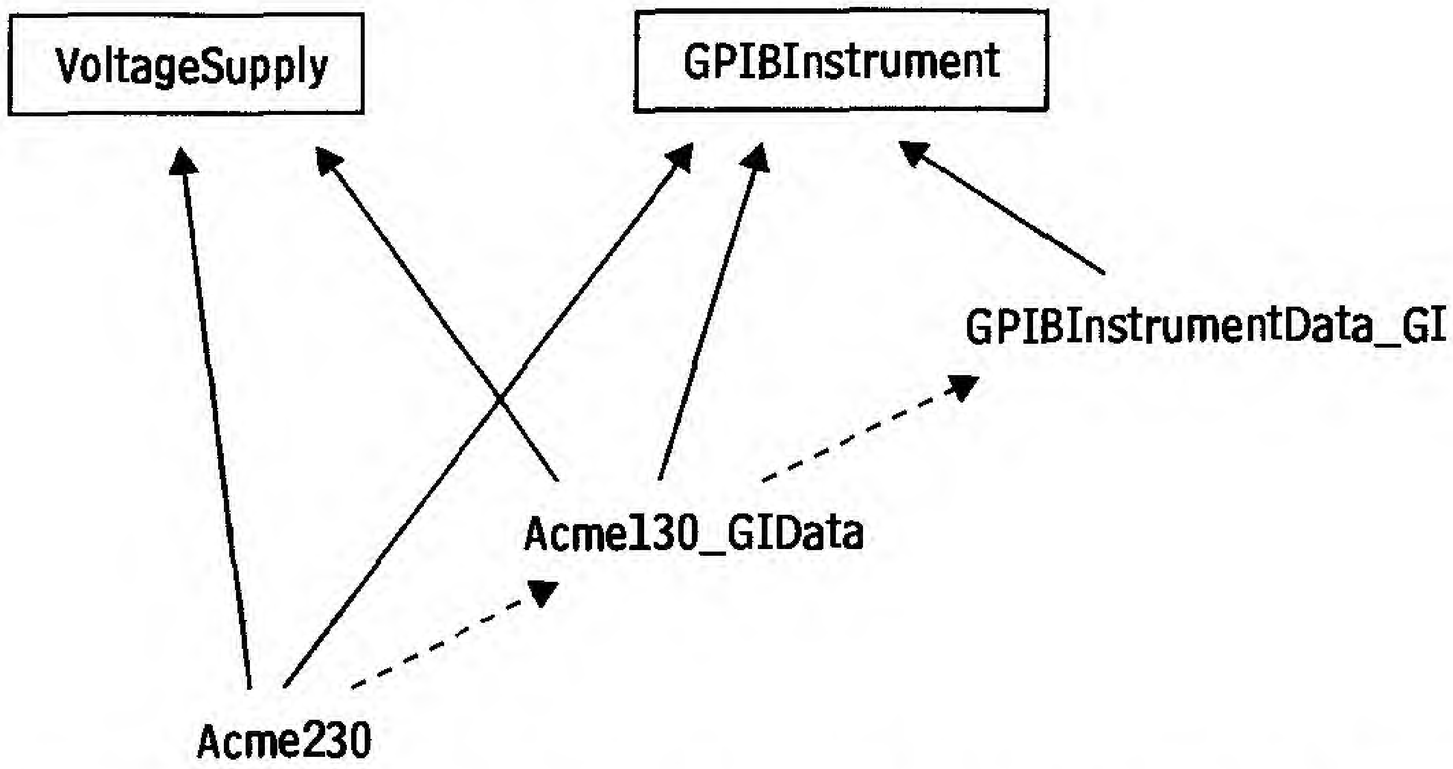


Figure 10.8 Class DAG for Acme230. The classes VoltageSupply and GPIBInstrument are interface base classes. GPIBInstrumentData implements GPIBInstrument and protectAcme130_GIData implements VoltageSupply. Dashed arrows indicate private derivation.

Notice that we now have two set() functions, one for setting voltage and one for setting voltage range. The class DAG for the Acme230 is shown in Fig. 10.8. The details of this class are explained in the following subsections.

10.6.1 Scope

As we discussed in Section 4.6, a class is a *scope* and the scope of each member function is nested in its class's scope. For example, Acme230's maximum() function uses the name high; according to the behavior of nested scopes, this name is interpreted as the high enum value in the scope Acme230:

ch10/Acme230.C

```
float Acme230::maximum() const {
    if (range_setting == high) return 100.0;
    return 10.0;
}
```

Names in class scope can be accessed by code outside of class scope using the scope resolution operator (::). For example, consider setting the voltage range for an Acme230 in some function not part of Acme230:

ch10/tAcme230.C

```
Acme230 supply_3(gpib, 15);
supply_3.set(Acme230::high);
```

The name high appears in the scope of Acme230; outside the scope of the class it can only be used by explicit qualification using Acme230:: preceding high.

Scope has a key role in the mechanism of derivation. From the vantage of the derived class, derivation introduces the names from all base class scopes into the

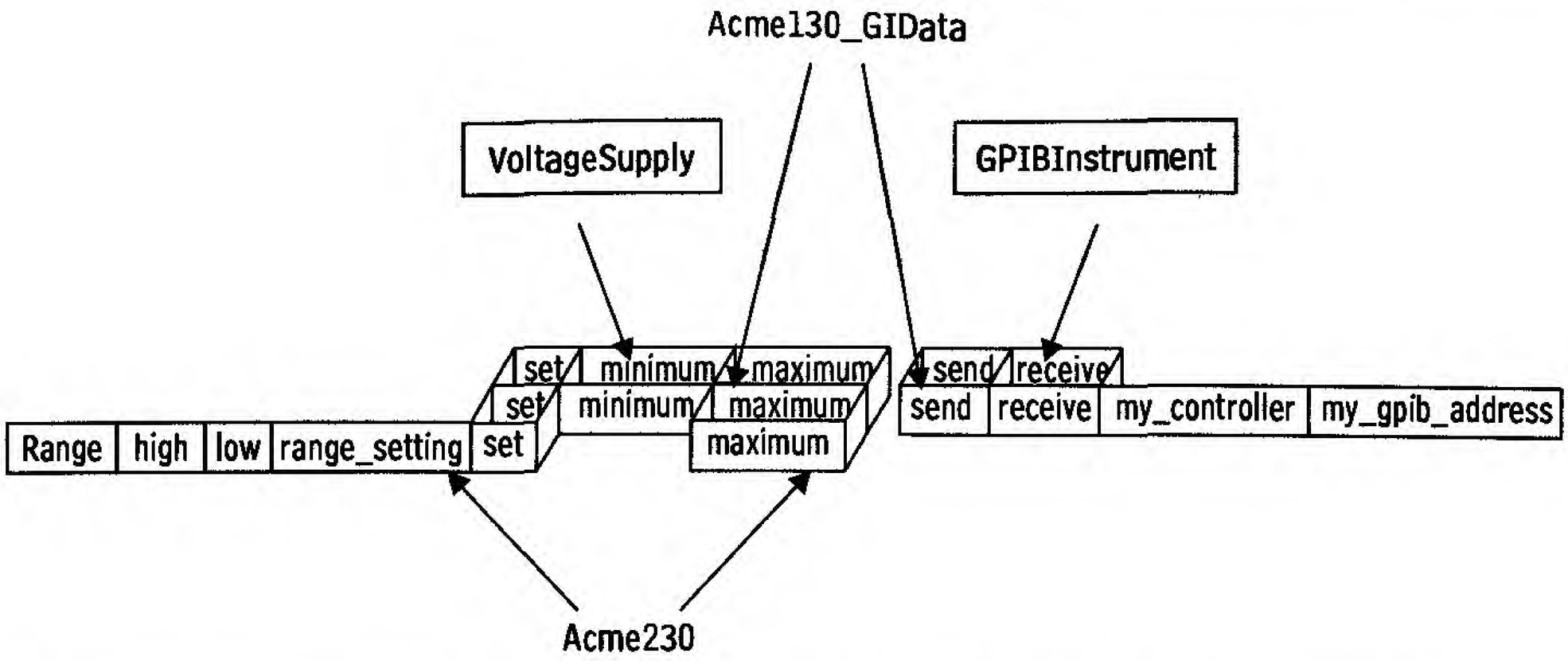


Figure 10.9 Name Hiding for Acme230. Names visible from the front correspond to names in Acme230.

scope of the derived class. Thus in the Acme230 class, the names declared in VoltageSupply, GPIBInstrument, and Acme130_GIData all appear in the scope of Acme230. In other words, names declared in base classes are *inherited* by a derived class.

The names in a derived-class scope are related to the names in a base class scope in a manner like the names in a block are related to names in an enclosing block. The inner block in each case can use a name declared in the outer block; a name in the inner block in each case has no impact on the names in the outer block. And in each case, a name in the inner block with the same identifier as a name in an outer block is said to hide the name in the outer block. For example, consider the declaration of maximum() in Acme230. Acme130_GIData already had a declaration for maximum, and Acme230 inherited the name. By redeclaring that name, Acme230 hides the name it inherited from the base class. On the other hand, Acme230 inherits the name minimum from Acme130_GIData, but does not hide it. Figure 10.9 illustrates name hiding for Acme230.

Name hiding applies to all names, but overriding also comes into play with virtual functions. When a derived class declares a virtual function having the same name as a virtual function in one of its bases, the corresponding base class name is hidden, as usual; if the function declaration matches the base class version in number and types of arguments, the base class version is also overridden. If the arguments don't match, then only hiding occurs and there is no override. Such a program might be correct technically, but such cases are almost always programmer errors so most compilers issue a warning.

Name hiding affected how we wrote Acme230. An Acme230 has two kinds of set behavior: The voltage can be set and the range can be set. We'd like to overload set(), with voltage setting and range setting distinguished by the argument type: float for voltage and Range for range. Voltage-setting behavior is already provided

by Acme130_GIData, and Acme230 could add range-setting behavior. It might therefore seem natural to inherit voltage-setting behavior from Acme130_GIData and add range-setting behavior with an additional set() member in Acme230.

§13.1 But name hiding occurs independently of and before function overloading. Therefore adding a new member named set to Acme230 hides the set inherited from Acme130_GIData, regardless of the argument types of the two set() functions.

To obtain two set() functions in Acme230, we must declare two set() member functions in Acme230, one to *unhide* the inherited voltage-setting behavior and one to add the range-setting behavior.

- To overload an inherited function, add the additional function(s) in the derived class and unhide the inherited function(s) of the same name.

To unhide the inherited voltage-setting function, we override it with a function that simply calls the base class function, using scope resolution:

ch10/Acme230.C

```
void Acme230::set(float voltage) {
    Acme130_GIData::set(voltage);
}
```

Calling a function using scope resolution suppresses the virtual function call mechanism, and the call is directed to the set() member of Acme130_GIData.

10.6.2 Dominance

Class derivation builds up a class starting with the declarations already present in the base classes. The derived class's scope is nested in the scopes of all of its base classes. As long as every name is inherited from one base, the derived class contains the names inherited from all its bases. If two base classes contain the same name, then either one name must dominate the other or the derived class must disambiguate the name by hiding the base class declarations.

Dominance is name hiding generalized to a full class DAG. A name declared in a derived class hides the same name declared in a single base class. A name declared in a derived class dominates the same name declared in any of its multiple bases. For a single base class, hiding and dominance are the same; for multiple base classes, dominance is the general rule.

Consider the class DAG for Acme230, Fig. 10.8, and focus on the classes derived from VoltageSupply. The declaration of minimum() in Acme130_GIData (page 281) dominates the declaration of minimum() in VoltageSupply. The name in the derived class also dominates the base class name in any DAG composed using this pair of classes. Thus when Acme230 derives from both VoltageSupply and from Acme130_GIData, the name minimum() in Acme130_GIData dominates the name minimum() in

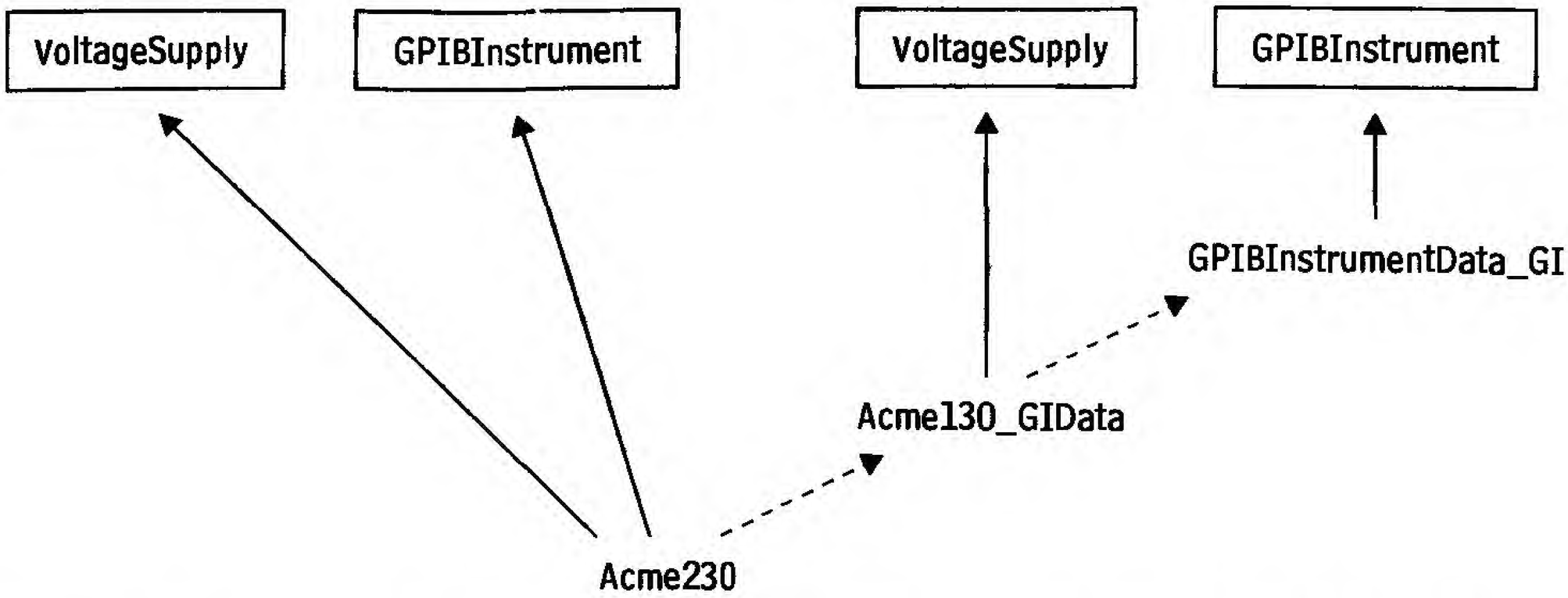


Figure 10.10 Class DAG for Acme230 if the virtual Keyword Was Omitted from the Interface Base Class Specifications. This class DAG is an incorrect representation of an Acme230.

VoltageSupply. Two base classes of Acme230 declare the same name, but Acme130_GIData derives from VoltageSupply so its name dominates.

Since dominance is defined in terms of names in a DAG, changing the structure of the DAG can change which name is dominant. Consider, for example, the DAG that would result if we removed the virtual keyword from the interface bases in the Acme230 DAG shown in Fig. 10.8. In that case, the interfaces declared for Acme230 would be distinct from those declared for the Acme130_GIData and GPIBInstrumentData base classes, as shown in Fig. 10.10. With this DAG, the minimum() name declared in Acme130_GIData still hides the name in its VoltageSupply base class, but it does not dominate the minimum() name declared in Acme230's second VoltageSupply base. The name minimum would be ambiguous and the C++ compiler would not be able to determine the meaning of minimum() in the scope of Acme230.

Returning to the correct DAG for Acme230 in Fig. 10.8, we can also think of dominance as the rule determining the meaning of an identifier. Thus send() used in the scope of Acme230 means GPIBInstrumentData_GI::send() because that name dominates the name in the GPIBInstrument base. Notice that proximity in the DAG is irrelevant: Acme230 is derived directly from GPIBInstrument and only indirectly from GPIBInstrumentData_GI, but that does not give names declared in GPIBInstrument any special significance compared to names inherited from other bases. Dominance, not proximity, determines the meaning of names.

Consistent application of the guidelines we have given in this chapter will cause implementation base class names to dominate interface base class names without further consideration. Compile-time ambiguity of names in a class DAG will then be limited cases of accidental *name collision*. When names from two different base classes are accidentally the same, we must either change one of the

two ambiguous names or redeclare the name in the derived class to *disambiguate* the name.

10.6.3 Member Access Control

We have already seen many examples of the use of the public and private access specifications to control access to data and function members. When a function uses a variable or calls a function, the C++ compiler first resolves the variable or function name within the current scope, finding the dominant name. In the case of function calls, overloading resolution is then applied (see Section 5.5) to select one function from among all functions in the current scope having the same name. For nonfunctions, the meaning of the name must be unique. Finally, after the meaning of the name is established, a check is done to ensure that the access is allowed. C++ provides three access specifiers—public, protected, and private—each applicable to both members and bases.

Access to a class member is determined by the access declared for it. Access to a base class member from a derived class is determined by a combination of the access specified by the base class for that member and the access specified in the derivation. The basic rule is that the access specified in the derivation can reduce but not increase the level of access specified by the base; see Table 10.1. To put it another way, once a name is encapsulated, derivation cannot break in.

public Members We use objects by calling their public member functions. The public members of an object's public interface base classes act like standard connections. Using standard connections makes it easier to upgrade and adapt a program. If Acme tried to sell its model 130 power supply with instructions to open the case and solder wires to points "J12" and "J9" for the output voltages, the price or performance advantages of the supply would have to be balanced against its

Base Class Access Specifier			
Base Member Access Specifier	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	Not accessible	Not accessible	Not accessible

Table 10.1 Access to a Base Class Member from a Derived Class. For base class member access declared as shown in the left column, the members assume the access in the derived class as shown in the right three columns.

nonstandard connections. Similarly, when we write Acme130, the member functions for setting voltage should be in VoltageSupply unless there are great advantages of not doing so.

- Consider every public member function a candidate for inclusion in an interface base class.

To put it another way, avoid writing public member functions that expose the internal representation of the class. Most public functions in most classes should be declared so that the function could be implemented by other similar classes. As the design evolves and these similar classes emerge, the category of classes sharing these function declarations can be formed by deriving from an interface base class.

The principal exceptions are the public functions of classes designed for efficiency over flexibility. Be especially suspicious of one-of-a-kind functions added to derived classes. In design iterations on such classes, try to promote these functions into interface bases to avoid building systems that are dependent on one class implementation.

private Members Encapsulation keeps objects from one class from altering the state of objects in another class. For safety, we do not normally run our voltage supplies with the case open. Similarly, we encapsulate Acme130 to prevent accidental access to its member data. Encapsulation does not prevent runtime accidents or provide runtime data security. Encapsulation operates at compile time.

Encapsulating an object's components restricts access to them, thereby preventing other classes from being written based on the details of the object's class. Providing member functions to manipulate the components in an object gives a class-controlled path to the details; formulating those member functions as standard interface components creates modular software. Encapsulation supports modularity.

A class's own private functions manipulate its private member data in ways too implementation specific to be made available via the public interface. Use private member functions to shorten long, complex public member functions into manageable units. However, a growing list of private member functions signals a need to split off member data and create a new class:

- Try to replace groups of private member functions with new private implementation base classes or private data members.

protected Members Between private and public members and bases lie protected members and bases. The specification protected means public access for derived classes and private access for others. Members of a derived class can call (or access) a base's public or protected members.

There are few compelling reasons to grant more access to derived classes than to other classes. Permitting access to implementation-specific members reduces modularity, regardless of the derivation relationship. Generally we recommend the following:

- Avoid protected members and bases.

There are two major exceptions to this guideline. First, it sometimes pays to revise code with execution-time bottlenecks at public member function calls to use protected member functions or to access protected member data directly. For example, a protected version of a function might assume that its arguments are correct, whereas the public version might check.

Second, it is often useful to hold implementation base class constructors protected, thus allowing derived classes to initialize their base subobjects while preventing non-class clients from creating instances. An example of this usage appears on page 376.

10.6.4 Direct Calls to Interfaced Private Base Class Functions

Implementating public interface functions by a private base class encapsulates the implementation while allowing calls through the interface. Thus the following code compiles and runs as expected:

ch10/tAcme130_GIData.C

```
GPIBController_GC gpib;

GPIBInstrument* slp = new Acme130_GIData(gpib, 12);
slp->send(15);
GPIBInstrument& supply1 = *slp;
supply1.send(15);
```

The member function send() is being invoked through the GPIBInstrument interface in both cases.

However, the following similar code does not compile:

ch10/tAcme130_GIData.C

```
Acme130_GIData supply1(gpib, 12);
supply1.send(15);    // WRONG: can't access private member
```

Here we are trying to call send() directly as a member of Acme130_GIData rather than through the interface. Members of a private base class are inherited as private members of the derived class. The member function send() is a public member of the GPIBInstrument interface, but a private member of Acme130_GIData, and therefore cannot be called by clients of Acme130_GIData. In many cases this behavior can help us create better programs:

- Using private inheritance to implement an interface forces the derived object to be used through the interface.

In other cases we wish to call the interface functions directly on the object. This can be achieved using access declarations, which we now describe.

10.6.5 Access Declarations

We have just seen that using private inheritance for implementation results in functions that are accessible through an interface but not through the object itself: private derivation turned a public member of an interface base into a private member of the derived class. C++ provides *access declarations* to adjust the access of inherited base class members. ARM §

An access declaration consists of the name of the base class, followed by the scope resolution operator (::), followed by the name of the member; it adjusts the access of the named member to public or protected, depending on where it appears in the derived-class definition. For example, here is a new version of Acme130 that uses access declarations to restore public access to send and receive:

ch10/Acme130_Acs.h

```
class Acme130_Acs :
    public virtual VoltageSupply,
    public virtual GPIBInstrument,
    private GPIBInstrumentData_GI {
public:
    Acme130_Acs(GPIBController& controller, int gpib_address);

// VoltageSupply interface
    virtual void set(float volts);
    virtual float minimum() const;
    virtual float maximum() const;

// GPIBInstrument interface
    GPIBInstrumentData_GI::send;
    GPIBInstrumentData_GI::receive;
};
```

We call this version Acme130_Acs, meaning "with access." With Acme130_Acs, the code (cf. page 292)

ch10/tAcme130_Acs.C

```
Acme130_Acs supply(gpib, 12);
supply.send(15);
```

compiles.

Access declarations grant access to names, not to individual functions. Thus access declarations do not specify function argument or return types, and access is granted to all members with the specified name. The access declaration for set in Acme130_Acs grants access to both of GPIBInstrumentData_GI's send() member functions.

Names encapsulated by a base class, like my_controller in the GPIBInstrument-Data_GI base class, cannot be made public in a derived class. Such names are not encapsulated by the derived class but by the base class. Contrast this to names like send that are not encapsulated by the base class; only the private derivation of Acme130_GIData from GPIBInstrumentData_GI caused send to be encapsulated. Acme130_Acs undid this with an access declaration. Access declarations cannot, however, restrict access to an accessible member of a base class. For example, an access declaration cannot make a protected or private member of a public member of a public base class.

Composing public interface base classes with private implementation base classes, supplemented with access declarations to bring functions of the private base class public, is a convenient and robust way to build systems of classes. In the next section, we review this important technique.

10.7 Base Class Composition

In Chapter 9, we introduced virtual functions in interface bases. We then derived classes from these interface bases and implemented the virtual functions there. In this chapter, we discussed implementing the virtual functions of an interface base by private derivation using the virtual base mechanism to connect the interface of the derived class to the interface of the implementation base class.

C++ does not require this particular style of programming with virtual functions. Neither the separation of the virtual functions into an interface base class (separate from any member data) nor the use of private implementation base classes with virtual public interface base classes is required to use virtual functions. To illustrate the advantages of the style we use here, a style we call *base class composition*, let's look at some alternatives.

We could, for example, have used public derivation to build Acme230 from Acme130_Acs. Then we would not need virtual base classes in Acme130_Acs. Using the suffix NoV for "no virtual bases," our classes might have been defined like this:

ch10/tNoV.C

```
class Acme130_NoV :
    public VoltageSupply,
    public GPIBInstrument {
    // members here...
};
```

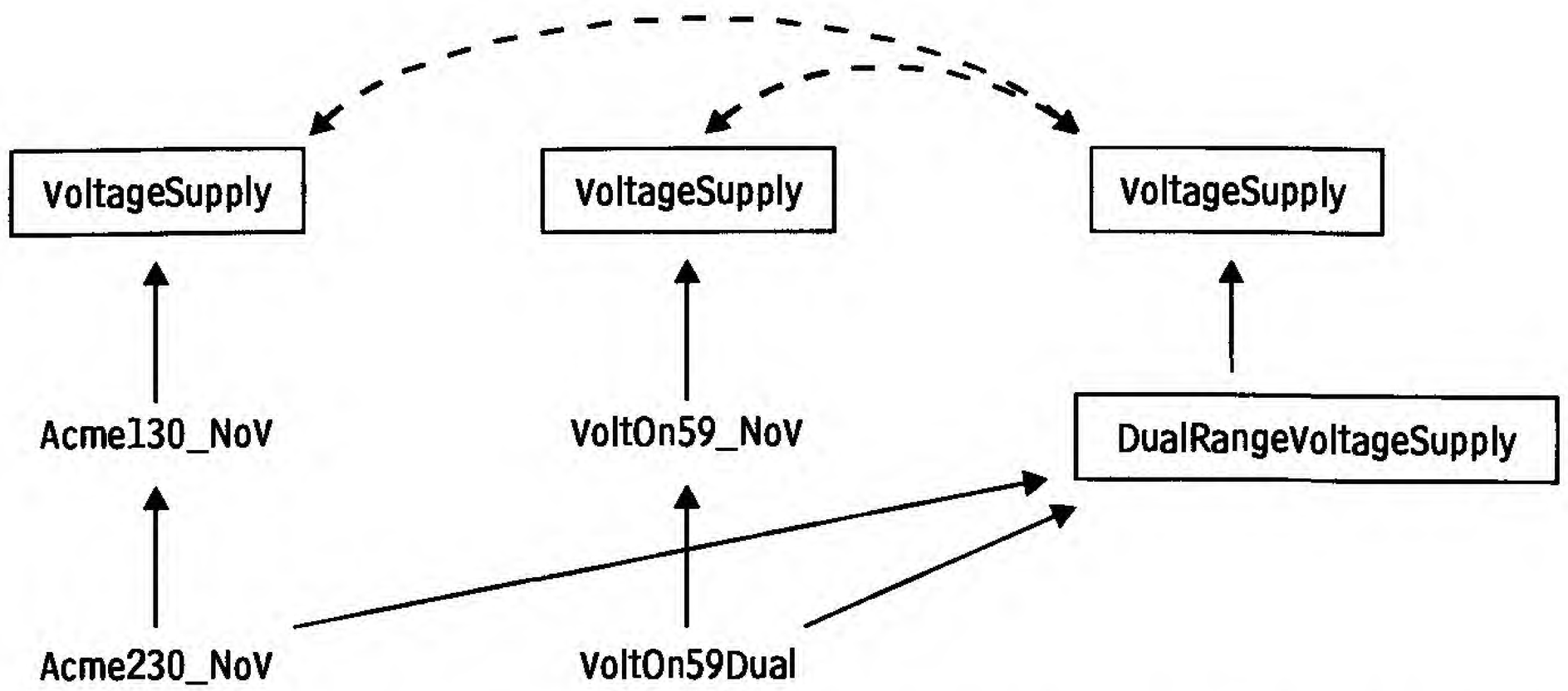


Figure 10.11 A class DAG for Voltage Supplies Built Without virtual Base Classes and Using public Inheritance of Implementation. A new interface to voltage supplies, DualRangeVoltageSupply, that adds set(Range) to VoltageSupply could not be added to this DAG. The two VoltageSupply base classes would conflict, as indicated by the dashed double arrows.

```
class Acme230_NoV :
    public Acme130_NoV {
    // members here...
};
```

giving the class DAG shown on the left-hand side of Fig. 10.11.

The drawbacks to this approach appear when the system grows. Suppose that to compete with the Acme company, the VoltOn company develops its own dual-range voltage supply, calling it the VoltOn 59 "Dual." We could add a VoltOn59Dual class to our system, following the pattern of the Acme230_NoV, to give the classes in the center column of Fig. 10.11. But we would not be able to extend the system to introduce an interface for all dual-range voltage supplies. As shown in the left-hand column of Fig. 10.11, deriving Acme230_NoV and VoltOn59Dual from DualRangeVoltageSupply would create ambiguous VoltageSupply base classes.

Introducing virtual interface bases transforms the DAG of Fig. 10.11 into the DAG of Fig. 10.12. There is no ambiguity here. The virtual specifier for interface base classes allows new interfaces to build on existing interfaces without conflict.

Although better than Fig. 10.11, the DAG in Fig. 10.12 does not encapsulate the implementation of either Acme230 or VoltOn59Dual. Thus the compiler will not help remind us to write client functions using the interfaces. Using Acme230 objects as Acme130_Acs objects will be fine until the day that we change the implementation of Acme230. On that day, all such uses of Acme230 will be wrong. If private implementation inheritance had been used, as shown in Figure 10.13, Acme230 and

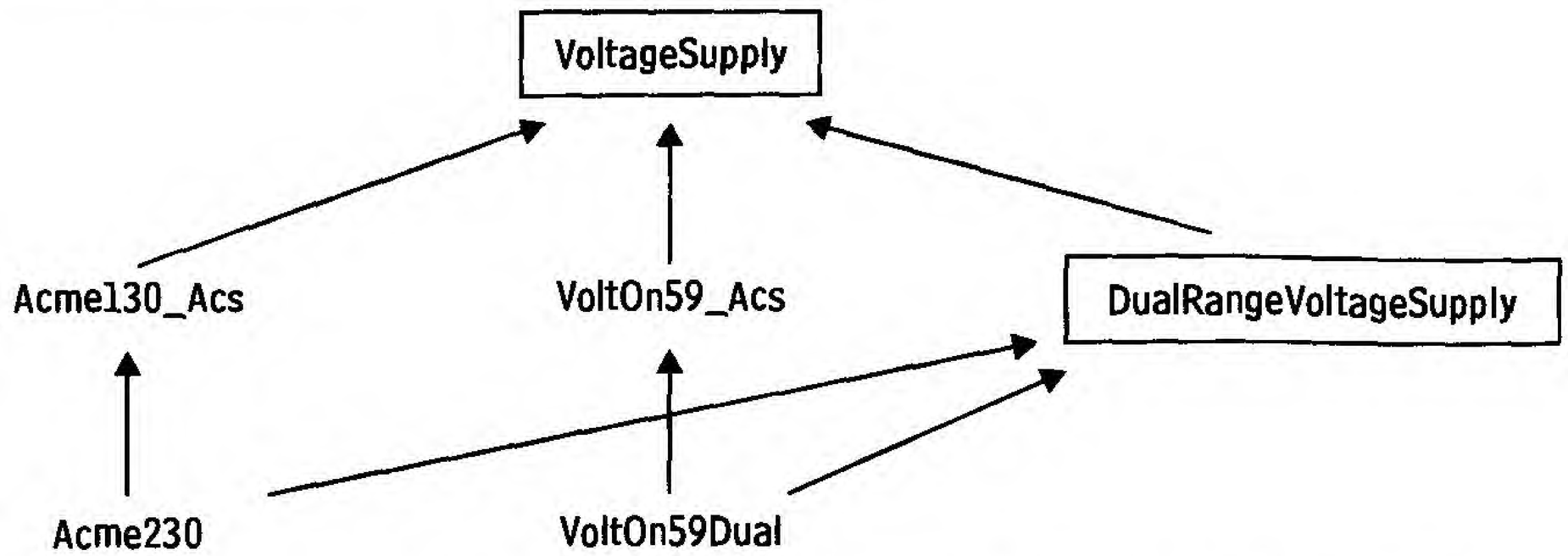


Figure 10.12 A Revision of Fig. 10.11 That Uses Virtual Base Classes for Interfaces.

VoltOn59Dual objects would always be used through interfaces. Notice that this diagram is structurally the same as Fig. 10.12. Only the access has changed.

In even larger systems, a pattern of base class composition emerges. Classes that extend the behavior of an existing class derive from interfaces that extend the interfaces of the existing class. For example, Acme230 extends the behavior of Acme130_Acs, and Acme230's DualRangeVoltageSupply interface extends Acme130_Acs's VoltageSupply interface. This creates a diamond-shape DAG, as illustrated for Acme230 alone in Fig. 10.14: The original interface is on top, the interface extension on one leg, the implementation of the interface along the other, and finally the implementation of the extension along the bottom. This *virtual base diamond* structure is characteristic of composition when interface and implementation are separated. Cascaded use of the virtual base diamond pattern makes it easy to grow the function of a system of classes while maintaining the separation of implementation and interface. However, today's compilers introduce overhead for each use

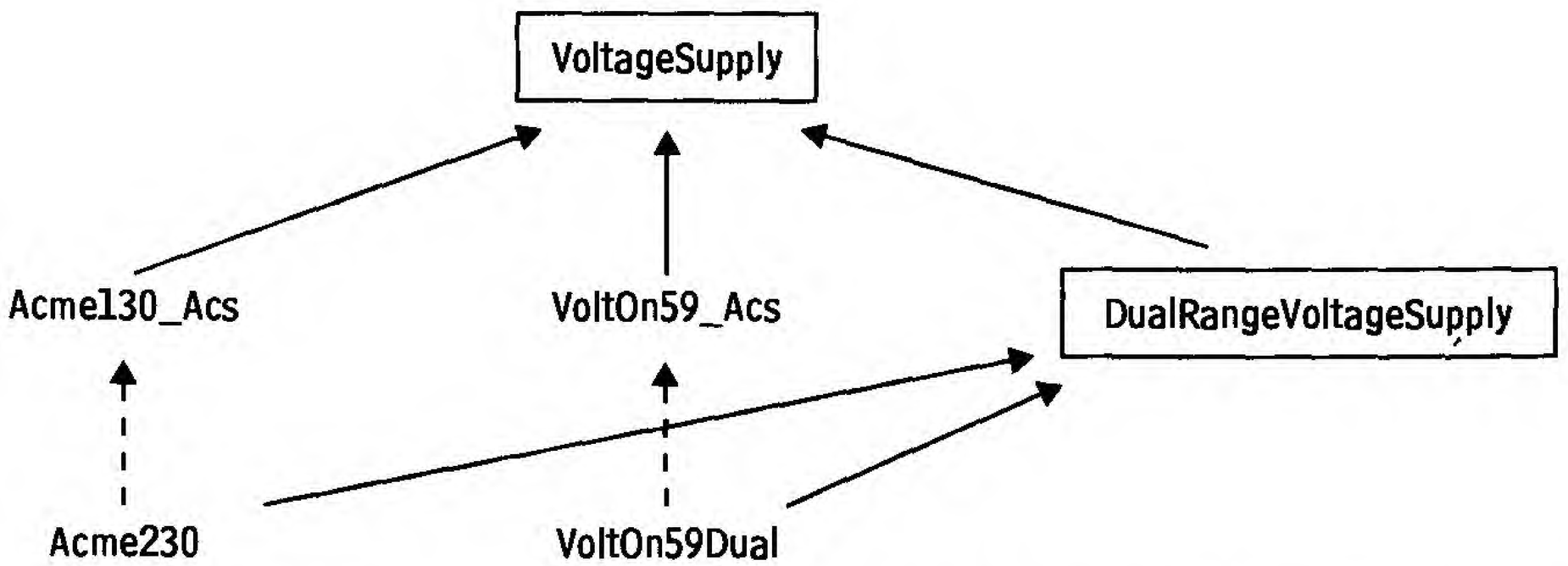


Figure 10.13 A Revision of Fig. 10.12 That Uses Both Private Implementation Inheritance and Virtual Base Classes for Interfaces.

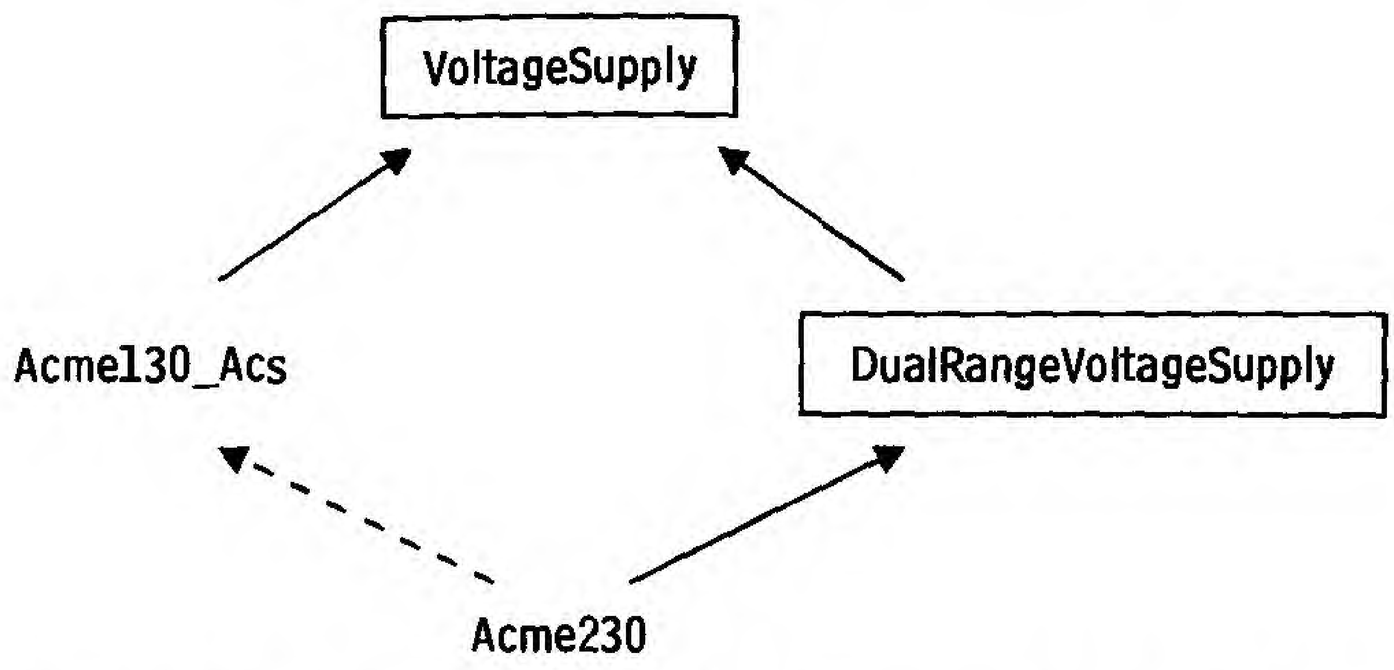


Figure 10.14 DAG for Acme230 Illustrating the virtual Base Diamond Pattern.

of virtual bases, so you must balance flexibility and efficiency; see Notes and Comments 10.5.

10.8 The Meaning of Declarations in a Class

We can now summarize the relation between the function declarations in class bodies and the functions available to act on objects. A function declaration in a class body exists for one of the following purposes:

1. To introduce a new function applicable to the class;
2. To hide a base class function name in the scope of the class;
3. To unhide a base class function hidden by another function of the same name;
4. To disambiguate functions with the same name from different base classes; and
5. To override a virtual function.

The complete set of functions available for a class can be determined by combining its explicit declarations with the names introduced by derivation from base classes according to the rules of scope, dominance, and access.

To know all the functions applicable to objects in a derived class, we must study the class and all of its base classes. This work increases with the length of paths in the class DAG, which argues for "shallow" DAGs. On the other hand, "deep" DAGs allow more sophisticated and reusable classes to be constructed. We choose to

- Design deep interface category DAGs and shallow implementation DAGs.

By this criterion, we accept the complications of a deep lattice for interfaces but minimize the complications on the less vital implementation side. We also note that as programming environments begin to provide better tools for working with C++, the problems of working with deep class DAGs should decrease.

10.9 Summary

C++ uses inheritance to express common implementation. Common implementation members are placed in a base class; every class sharing the implementation derives from the base class. A derived class inherits the members—both data members and function members—of its base classes; a derived class can declare new members with new names or with the same name as a member of one or more of its base classes. The final result, the combination of inheritance from base classes and declarations in the derived class, describes the class.

An *implementation category* is a group of classes that are all derived from the same base class for the purpose of expressing common implementation. In Section 10.4, we created an implementation category by extracting `GPIBInstrumentData` from the implementation of the GPIB classes in Chapter 9. This design expresses the commonality among objects that represent GPIB-connected instruments—they all have `GPIBInstrumentData`—helping to keep our C++ code close to our mental image. There is also a more tangible benefit to using implementation categories: The member functions of the base class apply directly to all classes in the category. For example, code to implement the `GPIBInstrument` commands can be written once, in `GPIBInstrumentData`, and then used from any class in the category.

Both interface and implementation categories traffic in member functions, but for different ends. An interface category requires its classes to provide definitions that adhere to a common concept so that code outside the category written to use the interface functions will work for all of the classes. An implementation category provides common definitions to classes derived from it.

The example classes in this chapter inherit only a small amount of code. The benefits of code sharing are proportionally larger with larger, more realistic classes. However, we only arrive at larger, well-structured systems by consistent use of class derivation through several design iterations that begin with simple classes. We recommend that you

- Apply class derivation whenever possible.

Occasionally this results in too many layers of classes and some trivial inheritance, but this is preferable to an incoherent group of classes. Recognize class derivation as a fundamental tool in C++.

We reused the code in `GPIBInstrumentData` in three different ways: public derivation from `Acme130_VS_GI_GC` (page 271), member function forwarding to a pri-

vate GPIBInstrumentData member (page 278), and private GPIBInstrumentData base with access declarations (page 293). Which solution is best?

- Use direct public derivation to try ideas for simple systems and for simple extensions. This method is the easiest one to apply but exposes the use of the base class as part of the implementation of the derived class.
- Use private derivation with access declarations as a more robust substitute for public derivation in systems that you expect to evolve.
- Use member function forwarding when you want most of the maintenance benefits of private derivation with access declarations but can't afford the overhead of virtual bases introduced by present-day compilers.

Inheritance can be used for one-of-a-kind class extension. Using direct public inheritance requires the programmer only to recognize that the problem at hand—the design of the Acme140 class, for example—is partly solved by the existing Acme130 (base) class. View inheritance as extension if you have a class system to exploit. More generally, inheritance expresses implementation commonality; view it as a mechanism for expressing commonality if you are building a class system.

10.10 Notes and Comments

10.1 A virtual function call has more overhead than a non-virtual function call: Measurements made by Gorlen *et al.* [51, Section 9.2], on one particular compiler, show a factor of 2. In our experience, the factor is less. Regardless, we have seen few situations in which the additional overhead of a virtual function call is important when compared to the overhead of a non-virtual function call.

10.2 With present-day compilers, declaring a virtual function to be inline rarely improves execution speed and almost always increases code size. A compiler can only generate inline code when it knows the actual type of the objects the function manipulates; in other words, the function must access the objects directly as variables, not via pointers or references. However, objects with virtual functions are most often accessed via pointers or references. See Item 33 of [82] for further discussion of how most compilers treat functions that are declared inline but for which the compiler can't generate inline code.

10.3 Using public derivation to express implementation always violates the separation of specification and implementation, preventing encapsulation. However, real programs must get written even if it means a mix of specification and implementation considerations. Determining which functions and members belong to a category of classes and which belong to a single class cannot always be done when we have only a single class or two classes whose relation we have not yet nailed down. Only iteration of both

specification and implementation can improve these choices: Using public inheritance of implementation often makes sense at the beginning of a project.

10.4 Excellent discussions of encapsulation in object-oriented languages, with particular emphasis on the interaction between encapsulation and inheritance, appear in [100] and [101]. The design of C++'s encapsulation mechanisms is discussed in [44, Chapter 11].

10.5 Deriving from more than one base is called *multiple derivation*, whereas deriving from only one base is called *single derivation*. Early versions of C++ did not allow multiple derivation, and its utility has been questioned.

Cargill [21] has argued that the benefits of using multiple derivation do not repay the extra complexity it introduces into C++ and C++ compilers. The crux of his argument is that member function forwarding can be used in place of multiple derivation. Often this is true, but occasionally it is not. Regardless, forwarding functions are a maintenance burden.

Specifying unconnected ("orthogonal") behavior in separate interface classes allows more classes to share a given interface base class, leading to greater reuse [77]. This requires multiple derivation. Single derivation also encourages mixing interface and implementation, leading to inflexible, unadaptable base classes. Conversely, mixtures of interface and implementation make poor candidates for multiple derivation.

Many of the problems attributed to multiple derivation can be reduced with attention to separation of interface and implementation and a restricted style of using virtual bases. We describe these techniques in [85].

10.6 In the same vein, Meyers [82, p. 165] argues against extensive use of multiple derivation, and especially of virtual bases. He even refers to the "dreaded diamond-shaped inheritance graph." His arguments boil down to language complexity, inefficiency, and the difficulty of deciding when to use virtual derivation.

The complete rules for multiple derivation in general and virtual bases in particular are complex. We don't present them in this book and find little or no use for them in our programming. Most of the complexity relates to the rules for initializing and destroying virtual bases. We separate interface and implementation and use virtual bases for interfaces: Since they have no data and no constructors, the complex rules are irrelevant. All the destructors we write in interface bases have empty bodies. The upshot is that we avoid the complexity problems of virtual bases by using them in a restricted way.

Using virtual bases does impose both storage and runtime costs. We prefer to get our designs clean, with well-separated interface and implementation, before focusing on efficiency. If the resulting program is too inefficient, it can be transformed into a more efficient program, often by using member function forwarding. In [85], we have described an optimization technique that compilers could use to generate efficient code for the special case in which all virtual bases are interfaces (without data); however, we don't know of any existing compiler that does this.

We avoid the difficulty of deciding when to use virtual derivation with a simple rule: Interface bases should be virtual bases; other classes should not.

10.11 Exercises

10.1 As written, the CheckedFloatArray constructor doesn't check that the number of elements requested is reasonable. Add such a check and an exception object to be thrown when the error is detected. Make sure that your solution initializes the SimpleFloatArray base subobject correctly.

10.2 Use public inheritance from SimpleFloatArray to write an AveragingFloatArray class. Hide the base class subscript operator with a subscript operator that returns a float instead of a float& so that elements cannot be altered via subscripting. Add a setElement() member function for setting elements. Also add an average() member function that returns the average of the array's elements. Assume that calls to average() are both more frequent and more costly than changes to the elements, so use a data member to hold the current average.

10.3 Refer to Exercise 10.2. Write a function breakAverage(AveragingFloatArray&) such that passing an AveragingFloatArray object to breakAverage() causes the next call of average() for that object to return the wrong answer.

10.4 Implement a new VoltOn59 and VoltyMetrics class using member function forwarding to a GPIBInstrumentData member variable.

10.5 Reimplement CheckedFloatArray from Section 10.1 using member function forwarding to a private SimpleFloatArray rather than inheritance.

10.6 Implement a version of Acme140_Fwd that is functionally equivalent to the Acme140 on page 271 but that uses member function forwarding to implement both the VoltageSupply and GPIBInstrumentData interfaces.

10.7 Implement an Acme140 using private derivation from an Acme130_GIData. Discuss the advantages and disadvantages of your class compared with the Acme140 class in Section 10.1.

10.8 Reimplement CheckedFloatArray from Section 10.1 using a private base class SimpleFloatArray. Why is this case different from the reimplementation of Acme130_VS_GI into Acme130_GIData in Section 10.5?

10.9 For the Acme230 described in Section 10.6, list the scopes for each member function and member variable contained in the class or its bases. For each case give the access permission.

10.10 Reimplement Acme230 such that it still can be used as a VoltageSupply and as a GPIBInstrument, but use no virtual base classes.

CHAPTER 11

Expressing Common Structure

To WRITE maintainable code we cannot go about writing an array class for int objects, then one for double objects, then one for arrays of int objects and one for some new exotic object we need today. One approach is to rely on the C++ built-in arrays. But then we face the same problem when we want sparse arrays, lists, matrices, quaternions, and so on.

C++ provides templates to abstract the commonality inherent in concepts like arrays, lists, and so on. Having introduced templates in Part I, we use this chapter to analyze the role of templates and to discuss the interplay of templates, interfaces, and inheritance. We characterize template commonality as *name commonality* and *implementation structure commonality*. We introduce these ideas in this chapter, using the simple `SimpleArray<T>` template described in Section 4.3. We then discuss how templates can be combined with the advanced C++ features we covered in the two preceding chapters.

We also use this chapter to introduce the system of array classes that we develop in Chapter 13. Although this emphasizes arrays, we hope you will be able to see beyond them to the issues that we illustrate with this example.

11.1 Commonality Expressed by Templates

Class templates act almost like class code with compile-time variables in them: Pass in an actual argument for the variable and out comes new class code, with the actual argument substituted in the variable locations. The closest analog in C is the preprocessor macro; the closest analog in FORTRAN is hand duplication of code. The view of templates as generating code—macro expansion—helps in understanding simple applications like those in Section 4.3.

This chapter will work to go beyond this mechanical view by discussing how to recognize the commonality that leads to templates and by describing a variety of ways that templates interact with other C++ features. We aim toward a

view of templates as an expression of "has-structure-of" commonality with a different character than the "is-usable-as" or "has-base-of" commonality we have discussed in the preceding two chapters. Ultimately we come to express our algorithms and form our systems with templates directly, no longer imagining them expanded with concrete parameter types.

In this process, we do not want to lose sight of the underlying simplicity of the C++ template model. A C++ class template specifies a family of isomorphic (same structure) classes. C++ places few restrictions on template code, requiring only that once the template parameters are substituted with specific types, legal code results. This allows a wide variety of uses for class templates.

One important point of terminology to keep in mind: The language element that specifies the common structure is a template. If the structure is a class, it is a *class template*; if the structure is a function, it is a *function template*. Once the variable parameters have been fixed, we have a class or a function. To describe the origin of such a class or a function, we can say it is a *template class* or a *template function*, respectively. A template class results from fixing the parameters of a class template (See Fig. 11.1).

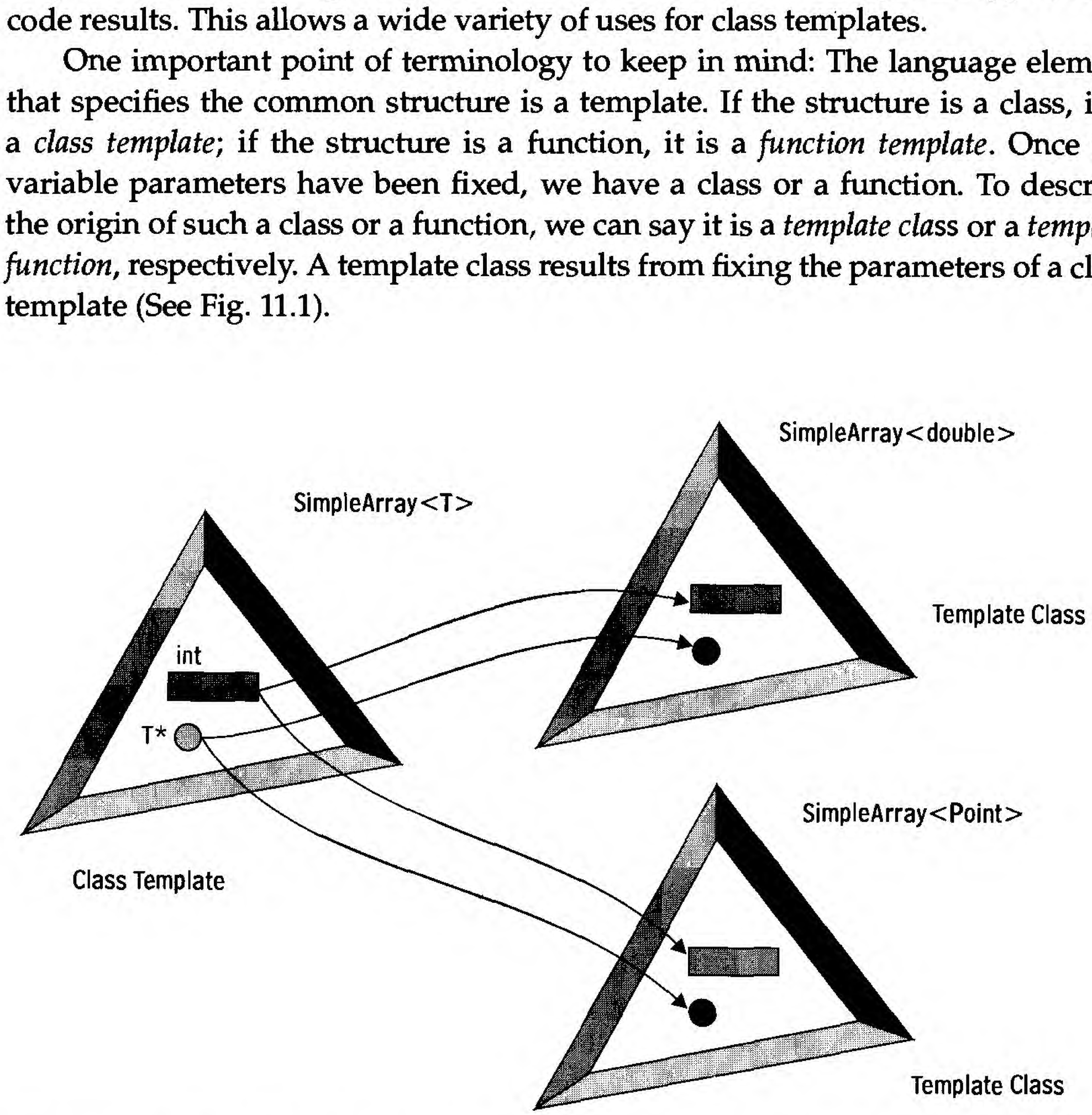


Figure 11.1 The Class Template `SimpleArray<T>` from Page 101 Declares an `int` Member and a `T*` Member. These are sketched inside the object on the left, using an outline for the `T*` member. When the template is expanded with specific types `T`, classes result as shown on the right-hand side. For `SimpleArray< int>`, the `T*` becomes an `int*`; for `SimpleArray< Point>`, the `T*` becomes a `Point*`. The class template itself is not a class.

The archetypal use of class templates is defining container classes, classes whose instances hold other objects. Arrays are the most basic example; other data structures such as linked lists and sets are also containers. The behavior of a container class is (largely) independent of the type of the objects its instances hold.

For example, the defining property of an array—constant time access to elements via subscripts—is independent of the type of the array's elements. If we write a class for arrays of int objects and one for arrays of Point objects, the member functions in these classes will likely have the same names and most of their code will be the same except for the element type. In this section, we use container classes to illustrate the two kinds of commonality that are expressed by templates.

11.1.1 Name Commonality

Consider the one-dimensional array class template SimpleArray<T> that we developed in Section 4.3 (page 101) and extended in Exercise 6.2 (173). All classes generated from the SimpleArray<T> template have members of the same names: You can call numElts() or use operator[]() or do assignment on a SimpleArray<int> or a SimpleArray<float>, or any other instance of this class template. This is an example of *name commonality*: The names are the same, but the argument and return types may depend on the value of the template parameter T.

Using these common names, function templates can act as clients of their class template arguments, leading to a single template function working on objects from many distinct classes. For example, the following function writes a comma-separated list of objects on an output stream:

ch11/tuplize.c

```
template<class T>
ostream& tuplize(ostream& os, const SimpleArray<T>& a) {
    // Output a comma-separated list of T's
    int n = a.numElts();
    for (int i = 0; i < n; i++) {
        os << a[i];
        if (i < n-1) os << ", ";
    }
    return os;
}
```

When we run the following code:

ch11/tTuplize.C

```
SimpleArray<int> ia(4);
ia = 3;
tuplize(cout, ia);
cout << endl;
```

```
SimpleArray<double> da(4);
da = 5.1;
tuplize(cout, da);
cout << endl;
```

we get

```
3, 3, 3, 3,
5.1, 5.1, 5.1, 5.1
```

A single tuplize() function template is defined and used for two distinct array classes.

In the preceding example, two different pairs of functions were called, the numElts() and operator[ ]() functions for SimpleArray<int> and the numElts() and operator[ ]() functions for SimpleArray<double>. A single declaration of SimpleArray<T> and a single definition of tuplize() were written: The common names of the functions expressed in the class template allow a single tuplize() function to work on all instances of SimpleArray<T>.

11.1.2 Implementation Commonality

Viewed from the standpoint of a class's clients, class templates provide name commonality: tuplize() relies on SimpleArray<T> to provide numElts() and operator[ ](). Viewed from the standpoint of a class's implementor, class templates provide *implementation structure commonality*: The definition of operator[ ]() for SimpleArray<T> on page 101 is valid for both SimpleArray<int> and SimpleArray<double>. One piece of source code supplies a common implementation.

The implementation commonality expressed in templates is not the same as that expressed in implementation base classes. If you have written the loop that copies the elements of an array of integers into the corresponding elements of another array of integers, the code will not copy elements of arrays of floating point numbers. But the structure of a loop suitable for int elements would be identical to the structure of a loop for double elements. If we abstract the element type in the loop to be a parameter, then the common structure of the loop implementation for copying elements can be used for both element types. Function templates for class members allow us to express such implementation structure commonality. Inheritance from an int array base class would reuse a loop over int elements; an array template allows us to reuse the structure of the loop independent of the element type.

For large systems, expressing this kind of commonality is vital. Each time a particular class is generated from a class template, the relevant member function definitions are generated from the corresponding function templates. Thus we don't have to write similar code over and over again; the compiler does that job

for us. More important, if we need to change the code, we do it in one place and the compiler replicates the change automatically.

11.2 Templates and Inheritance

In this section, we explore deriving class templates from implementation bases as a means of expressing commonality of implementation. We consider two examples. In the first example a class template is derived from a nontemplatized class; in the second example a class template is derived from another class template. Chapter 13 has examples of class templates derived from their template parameters, and Section 12.7 illustrates base class templates parameterized by derived classes.

11.2.1 Factoring Implementation from Template Categories

A class template specifies a category of classes sharing common names and common implementation structure. The classes in such a category may also share common member data or member functions independent of the template parameter. To express this dual commonality, we derive the class template from an implementation base class that is not a template.

For example, consider the Fallible<T> class template we developed in Section 6.4.4 (page 158). The member functions failed(), valid(), and invalidate() are independent of the type T, as are the member datum is_valid and the nested class UsedInInvalidStateErr (including its member functions and data). All of these are identical whether we have a Fallible<int> or a Fallible<double>.

We can factor these T-independent aspects of Fallible<T> into a nontemplatized implementation base class:

SciEng/Fallible.h

```
class FallibleBase {
public:
    FallibleBase(Boolean state) : is_valid(state) {}
    Boolean failed() const { return !is_valid;          }   // True if invalid.
    Boolean valid()  const { return is_valid;           }   // True if valid.
    void invalidate()      { is_valid = Boolean::false; }   // Make invalid.

    class UsedInInvalidStateErr :
        public SciEngErr {
    public:
        virtual String message() const;
    };
protected:
    void throwErr() const;
```

```
private:
    Boolean is_valid;
};
```

Then we derive from this implementation base in the T-dependent derived classes:

SciEng/Fallible.h

```
template<class T>
class Fallible :
    private FallibleBase {
public:
    Fallible(const T& t) : FallibleBase(Boolean::true), instance(t) {}   // Valid.
    Fallible()           : FallibleBase(Boolean::false)             {}   // Invalid.
    FallibleBase::failed;
    FallibleBase::valid;
    FallibleBase::invalidate;

    operator T() const;

    T elseDefaultTo(const T& default_value) const;    // Value if valid, else default_value
private:
    T instance;
};

template<class T>
inline
Fallible<T>::operator T() const {
    if (failed()) throwErr();
    return instance;
}

template<class T>
inline
T Fallible<T>::elseDefaultTo(const T& default_value) const {
    return valid() ? instance : default_value;
}
```

The factoring is apparent in the class DAG for template classes generated from the Fallible<T> class template shown in Fig. 11.2. Notice that we have also factored the original definition of the conversion operator (page 159) into a T-dependent part and a T-independent part (throwErr()) that throws an exception when necessary.

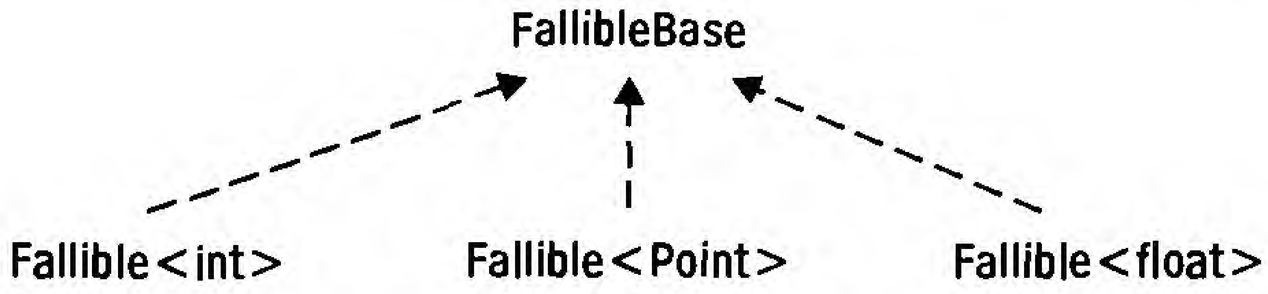


Figure 11.2 Class DAG for Some Template Classes Generated from Fallible<T> Class Template.

Sharing common implementation in this way reduces the number of functions created by template expansion. The member functions in FallibleBase are defined once, not written as function templates that are replicated for each type T for which the Fallible<T> template is expanded. For example, a single copy of these members will be compiled:

SciEng/Fallible.C

```
String FallibleBase::UsedInInvalidStateErr::message() const {
    return "Fallible object used in invalid state.";
}

void FallibleBase::throwErr() const {
    throw FallibleBase::UsedInInvalidStateErr();
}
```

Thus to avoid compiling replicated code,

- Place template parameter-invariant data and functions in implementation base classes and derive template classes from them.

We used private implementation inheritance here: FallibleBase is not likely to be useful on its own, and we do not wish to allow client functions to be written against this class. In the next example, a public extension view is more appropriate.

11.2.2 Extension by Public Inheritance

In Section 10.1, we described how to extend a concrete class using public inheritance. This technique generalizes straightforwardly to extending a class template by public inheritance. As an example, we'll rewrite the CheckedSimpleArray<T> of Section 4.5, this time using public inheritance instead of code replication.

Here is the class template definition:

ch11/CheckedSimpleArray.h

```
template<class T>
class CheckedSimpleArray :
    public SimpleArray<T> {
```

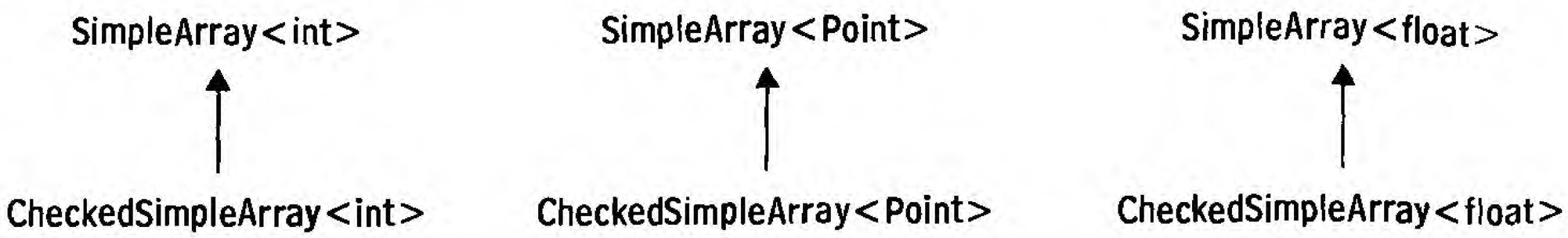


Figure 11.3 Class DAG for Some Template Classes Generated from the CheckedSimpleArray<T> Class Template.

```
public:
    CheckedSimpleArray(int n);                          // Create array of n elements
    CheckedSimpleArray();                               // Create array of 0 elements
    T& operator[ ](int i);                              // Checked subscripting
    CheckedSimpleArray<T> & operator=(const T&);        // Scalar assignment

    class SubscriptRangeError {};
};
```

Compare this with the definition of CheckedFloatArray on page 266. Notice that the base class is specified as SimpleArray<T>, not as SimpleArray, and that the return type of the assignment operator is parameterized similarly.

When the compiler generates a template class from the CheckedSimpleArray<T> class template, it also generates the corresponding base class, SimpleArray<T>, for the same T. Thus CheckedSimpleArray<int> is derived from SimpleArray<int> and CheckedSimpleArray<float> is derived from SimpleArray<float>, as illustrated in Fig. 11.3. There is no relationship between CheckedSimpleArray<float> and SimpleArray<int>. Contrast this DAG with the DAG shown in Fig. 11.2, in which the template classes are derived from a common base class.

The member function definitions also reflect the fact that each template class generated from CheckedSimpleArray<T> is derived from a class generated from the SimpleArray<T> class template for the same value of T:

ch11/CheckedSimpleArray.h

```
template<class T>
CheckedSimpleArray<T>::CheckedSimpleArray(int n) : SimpleArray<T>(n) {}

template<class T>
CheckedSimpleArray<T>::CheckedSimpleArray() {}

template<class T>
T& CheckedSimpleArray<T>::operator[ ](int i) {
    if (i < 0 || i >= numElts()) throw SubscriptRangeError();
    return SimpleArray<T>::operator[ ](i);
}
```

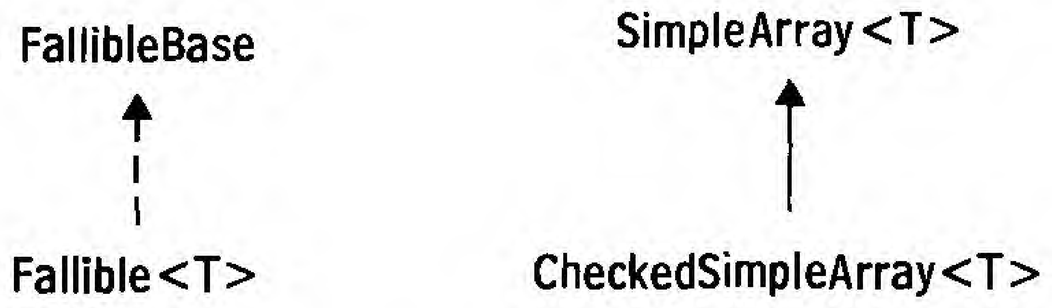


Figure 11.4 Illustration of Shorthand Notation for Class DAGs of Template Classes Generated from Class Templates.

```
template<class T>
CheckedSimpleArray<T>& CheckedSimpleArray<T>::operator=(const T& rhs) {
    SimpleArray<T>::operator=(rhs);
    return *this;
}
```

Every reference to the base class is parameterized by T; when these function templates are expanded, each resulting function refers to the appropriate base class member and function. If the base class member functions are defined by function templates, expansion of the derived-class functions automatically triggers expansion of the base class functions.

We have emphasized that deriving a class template from another class template establishes a derivation relationship between corresponding classes generated from the templates. For different values of the parameter T, there is no derivation relationship among the generated classes. Nevertheless it is often convenient when discussing and diagramming class DAGs to speak and draw as if there were a derivation relationship among the class templates themselves, bearing in mind that there is only a relationship among classes generated for a specific value of the template parameters. For example, as a shorthand we draw the class DAGs of Figs. 11.2 and 11.3 as shown in Fig. 11.4.

11.3 Example: Array Classes

Thus far we have concentrated on templates for concrete classes. In this section, we begin to look at interfaced classes. When we combine interface bases with class templates, we begin to exploit the full power of these features of C++. To illustrate templates and the combination of templates with interface and implementation bases, we need an example that consists of a system of interrelated classes.

This section introduces a family of array classes that we develop and use throughout the book. In this chapter, we focus on specific techniques for expressing commonality, not on the design and implementation of the array classes themselves; Chapter 13 takes the opposite view, focusing on building a consistent family of array classes using whatever techniques are appropriate. Although we will

not delve into the details of these arrays here, we will introduce the whole family to clarify the terminology and motivate the systematic treatment of array classes.

We adopt the following terminology: An *array* is a collection of *elements*, each an object of some fixed type, called the array's *element type*. An array may have one or more *dimensions*, and the number of dimensions is called the array's *dimensionality*. An element of a d-dimensional array is selected by providing d integer subscripts $s_0, \ldots, s_{d-1}$, with $0 \leq s_i < n_i$, where the integer n_i is called the *size* of the i-th dimension. The d sizes $n_0, \ldots, n_{d-1}$ comprise the array's *shape*.

ARM §8.2.4 C++'s built-in arrays are notoriously low level and are deficient in the following ways:

- Arrays are one dimensional, with multidimensional arrays represented as arrays of arrays.
- An array's shape is fixed at compile time. (As described in Section 2.13, the connection between arrays and pointers allows the shape of a one-dimensional array to be determined when the array is created. This technique requires attention to extraneous detail and does not generalize to multidimensional arrays.)
- An array doesn't know its dimensionality or shape. Therefore passing an array as a function argument requires passing its shape separately, resulting in unwieldy argument lists.
- Arrays do not behave like other objects: They are passed by reference, not by value, and they can't be returned by functions.
- Widely used array operations are missing. These include projection (e.g., selecting a row of a two-dimensional array), array assignment, and assignment of a scalar value to all the elements of an array.

In defining our own array classes, we seek to overcome these problems.

It is tempting to believe that one class template for multidimensional arrays could overcome these problems and meet most needs for scientific and engineering programming. However, different uses of arrays require different trade-offs among time efficiency, space efficiency, compatibility with existing code (e.g., FORTRAN subroutines), and flexibility. No single array class template can cope with the wide range of requirements and trade-offs that occur in practice [10].

We can characterize array classes by their flexibility in size and shape and by their storage format. We distinguish two levels of flexibility:

Rigid array. The dimensionality and shape of the array is fixed at compile time.

Rigid arrays are analogous to FORTRAN-77 arrays.

Formed array. The dimensionality of the array is fixed at compile time, but the shape of the array is determined at runtime when the array is created. The shape can be changed while the program executes.

Array elements can be stored contiguously, either in row major order, as in C and C++, or in column major order, as in FORTRAN, or non-contiguously, as in various sparse matrix representations, various packed representations that exploit matrix structure such as symmetry, or as in the scheme used in the book *Numerical Recipes in C* [93, Section 1.2].

We can also characterize array classes by their usability in client functions. We distinguish two different kinds of arrays based on the ways clients can use them:

Concrete array. The array class has no interface base class; new client functions must be used if new array classes are created. Client functions call array functions directly, with no virtual function call overhead.

Interfaced array. The array class shares interface base classes with other array classes; all classes with the same interface can be used in client functions accepting references or pointers to the base class.

Usability is orthogonal to flexibility: We can create concrete arrays and interfaced arrays for each flexibility. Like flexibility, usability has trade-offs: Concrete arrays will be more efficient in function-call time, and interfaced arrays will be more efficient in code space and programmer time.

A wide range of performance trade-offs is essential in practice. Interfaced, formed arrays will give the most adaptable program with the least amount of code; concrete, rigid arrays are essential for acceptable performance in programs with large numbers of small arrays. Moreover, as programs become more sophisticated, some parts need adaptability whereas other parts need time or space efficiency. Consider, for example, a program that manipulates 1000×1000 arrays containing space vectors, themselves three-element arrays. Any interfaced, formed array implementation will at least double the space required for the 3 vectors, but even a hundred words of space overhead would be insignificant in the arrays that contain the space vectors.

Our concrete array classes are elaborations on the idea behind `SimpleArray<T>`. They are described in Section 13.2. We use our interfaced array classes to illustrate the interactions of templates and interfaces in the next section.

11.4 Interfaced Array Classes

We design our interfaced array classes with one goal paramount: Any code that does not depend on flexibility or storage layout—code that depends only on "array-ness"—must work for every flexibility and storage layout, even those

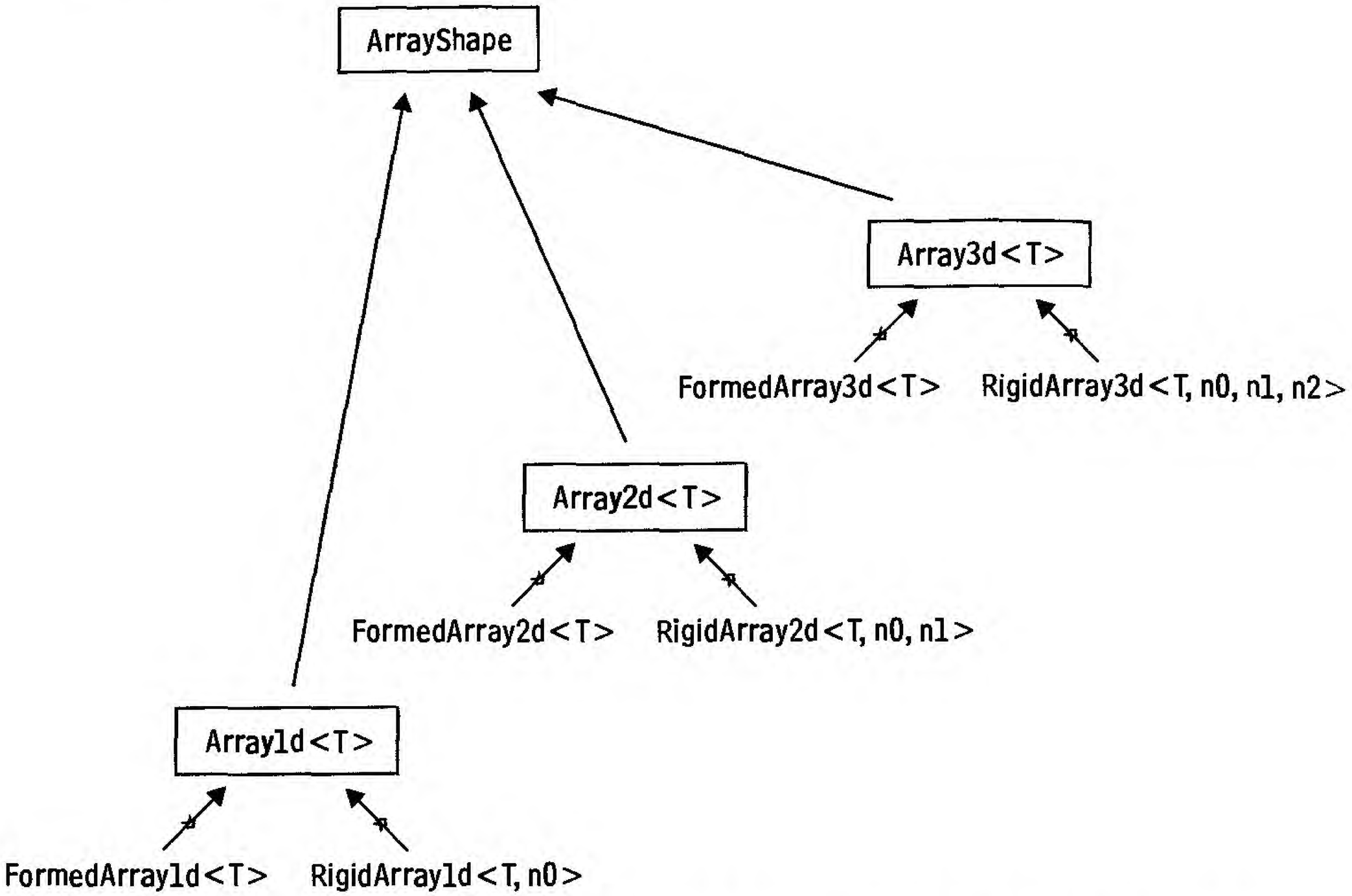


Figure 11.5 Class DAG for Family of Interfaced Array Class Templates. Interface bases are enclosed in rectangles. Broken arrows indicate derivation through classes not shown. Not shown are `FortranArray1d<T>`, `FortranArray2d<T>`, and so on, formed and rigid arrays with FORTRAN-compatible column-major storage layout; they are "leaf" classes like the unboxed names shown in this figure.

not yet written. In this section, we develop a system of templatized interface base classes for arrays. We shall rewrite these interfaces in Section 13.5 when we incorporate more of the features needed for fully usable arrays.

Figure 11.5 gives a partial road map of the array system. Starting at the top, `ArrayShape` is an interface for those aspects of arrays that are independent of both dimensionality and element type. The next layer of interfaces are dimension specific and parameterized by element type. At the leaves of the DAG are specific array implementations, all parameterized by element type—these are not interfaces. We adopt the names `RigidArray` and `FormedArray` for our implementations with the corresponding flexibilities. More precise names would be something like `RowMajorContiguousFormedArray`, rather than `FormedArray`. Each of these array implementations forms a template category over element type. Indeed our concept of "array" embodies the commonality of all arrays: We speak of an array meaning a multidimensional, multielement structure, with elements of identical but unspecified type. Thus an array type would be `FormedArray2d<float>` or `FormedArray2d<VoltageSupply*>`, both classes in template category `FormedArray2d<T>`.

11.4.1 Interface Common to Classes in a Template Category

The ArrayShape interface base class expresses what is common among all arrays regardless of dimensionality, element type, storage layout, or other implementation considerations. Once these issues are excluded, commonality boils down to representing dimensionality and shape. Our first task, then, is to decide how to represent dimensions and subscripts.

In our model of arrays, subscripts are nonnegative integers, and it is tempting to say that a subscript should be an unsigned int. But the range of an unsigned int is not necessarily equal to the number of elements in the largest possible array usable from C++. We could use size_t, an unsigned integral type defined by C++ with range large enough to hold the number of bytes in any object. However, as an unsigned type, size_t is prone to errors when used for loop indices. For example, the following loop prints the integers from 9 to 0 and then continues with large positive numbers: ARM §

ch11/sizet.C

```
for (size_t i = 9; i >= 0; i--) cout << i << endl;
```

Thus in practice subscripts may dip negative.

For this reason we use ptrdiff_t, a signed integral type that is the type of the result of subtracting two pointers to objects of the same type (i.e., of two pointers to elements of an array). It is defined in the standard header stddef.h. In typical implementations, ptrdiff_t and size_t have the same number of bits, implying that the maximum magnitude of a ptrdiff_t is about half that of a size_t. To us, the convenience of including negative values in subscript computations outweighs the reduced range. ARM §

Regardless, we use a typedef to provide a mnemonic name and to provide a single point of change when moving among machines with different word lengths:

SciEng/Subscript.h

```
#include <stddef.h>

typedef unsigned short int Dimension;
typedef ptrdiff_t          Subscript;    // Signed, machine-dependent int; from <stddef.h>
```

Once these two basic type decisions have been made, the interface is simple:

Array/ArrayShape.h

```
class ArrayShape {
public:
    virtual Dimension dim()                  const = 0;
    virtual Subscript  shape(Dimension d) const = 0;
    virtual Subscript  numElts()             const;

    virtual ~ArrayShape();
};
```

The dim() function returns the array's dimensionality, shape(d) returns the size in the *d*th dimension, and numElts() returns the total number of elements in the array.

The numElts() function can be written using only the functions in ArrayShape:

Array/ArrayShape.C

```
Subscript ArrayShape::numElts() const {
    Dimension d = dim() - 1;
    Subscript n = shape(d);
    while (d > 0) n *= shape(--d);
    return n;
}
```

This function is independent of the storage format or any other implementation details supplied by classes in the ArrayShape category: The interface guarantees that each nonabstract class in the category implements dim() and shape(). An implementation can override this function in the (unlikely) event that a more efficient implementation is required.

There is little else common among all arrays except exception classes. These are all independent of dimensionality or element type, and in an early version we nested them within the ArrayShape class. However, we want to share the exception classes between the interfaced array system we are developing here and the concrete array system we will develop in Chapter 13. Therefore we nest them in an ArrayErr class, used by the array classes but independent of them:

SciEng/ArrayErr.h

```
class ArrayErr {
public:
    class Dimensionality : public SciEngErr {
    public:
        virtual String message() const;
    };

    class Shape : public SciEngErr {
    public:
        virtual String message() const;
    };

    class CreationSize : public SciEngErr {
    public:
        virtual String message() const;
    };

    class NegativeSize : public SciEngErr {
    public:
        virtual String message() const;
    };
```

```
class Symmetry : public SciEngErr {
public:
    virtual String message() const;
};

class SubscriptRange : public SciEngErr {
public:
    virtual String message() const;
};
```

All of these classes adhere to the SciEngErr interface described in Section 9.10 and provide a simple error message:

ch11/ArrayErr.C

```
String ArrayErr::Shape::message() const {
    return "Array shapes not commensurate.";
}

String ArrayErr::Dimensionality::message() const {
    return "Array has wrong dimensionality.";
}

String ArrayErr::CreationSize::message() const {
    return "Creation size specified inconsistent with rigid array's shape.";
}

String ArrayErr::NegativeSize::message() const {
    return "Negative creation size specified.";
}

String ArrayErr::Symmetry::message() const {
    return "Asymmetric size specified for symmetric array.";
}

String ArrayErr::SubscriptRange::message() const {
    return "Subscript out of range.";
}
```

Exception objects in an industrial-strength array class library might store additional information used to create more detailed error messages.

11.4.2 Templatized Interface Categories

ArrayShape expresses commonality among all arrays regardless of element type or dimensionality. To define more specific interfaces, we have to tie down one or the other of these or both. Let's go through the thought process we used to arrive at the design shown in Fig. 11.5.

Suppose that we defined a dimensionality-independent interface parameterized by element type T. How could we use it? Not knowing the array's dimensionality, we couldn't access its elements via subscripts; we could only do whole-array operations like assignment, and even then we wouldn't be able to ensure at compile time that the two arrays being assigned have the same dimensionality. We therefore concluded that a dimensionality-independent interface wouldn't be useful.

The thinking for an interface independent of element type is similar. Element access by subscript depends on the type of the array's elements. For example, an array of float objects needs array subscripting that returns float objects, whereas an array of VoltageSupply* needs array subscripting that returns VoltageSupply* objects. We cannot create an interface base class that specifies subscripting without knowing element type. Whole-array operations like assignment aren't possible either because assigning (say) an array of circles to an array of squares doesn't make any sense. We therefore concluded that an interface independent of element type wouldn't be useful.

We need instead an interface base class that nails down both dimensionality and element type, something like this:

ch11/ArraynT.C

```
template<Dimension ndim, class T>
class Array :
    public virtual ArrayShape {
public:
    virtual Array<ndim, T>& operator=(const Array<ndim, T>& rhs) = 0;
    virtual Array<ndim, T>& operator=(const T& rhs) = 0;

    // virtual T& operator[ ]( ??? );   // Subscripting ??
    // ...
};
```

The interface base Array<2, float> would express the commonality among all two-dimensional arrays with float elements, regardless of implementation details like storage layout. When we try to write the declaration for the subscripting function, we run into trouble: We need to subscript with ndim arguments of type Subscript, but the square brackets operator, operator[ ](), used for subscripting built-in arrays

is restricted to one argument, and the number of arguments in template class member functions cannot depend on the template parameter.

We solve these problems by adopting the function call operator, operator()(), for subscripting. (We save the square brackets for projections; see Chapter 13). Then we define a separate class template for interface bases of a given dimensionality, encoding the dimensionality in the name and expressing the element type as a template parameter:

ch11/ArrayxD.h

```
template<class T>
class Array1d :
    public virtual ArrayShape {
public:
    virtual Array1d<T>& operator=(const Array1d<T>& rhs)       = 0;
    virtual Array1d<T>& operator=(const T& rhs)                = 0;

    virtual const T&    operator()(Subscript i)          const = 0;
    virtual T&          operator()(Subscript i)                = 0;
};

template<class T>
class Array2d :
    public virtual ArrayShape {
public:
    virtual Array2d<T>& operator=(const Array2d<T>& rhs)       = 0;
    virtual Array2d<T>& operator=(const T& rhs)                = 0;

    virtual const T&    operator()(Subscript i, Subscript j) const = 0;
    virtual T&          operator()(Subscript i, Subscript j)       = 0;
};
```

Each of these interfaces specifies the minimal level of functionality required for useful arrays of a given dimensionality.

For a given value of T, Array1d<T> and Array2d<T> are simply interface base classes. Client functions can be written to these interfaces, as usual:

ch11/ArrayxD.C

```
double sum(const Array2d<double>& a) {
    double sum = 0.0;
    for (Subscript i = a.shape(0)-1; i >= 0; i--) {
        for (Subscript j = a.shape(1)-1; j >= 0; j--) sum += a(i, j);
    }
    return sum;
}
```

This function will work for an instance of any class in the Array2d<double> category. Moreover, since exactly the same interface structure is used for array classes for all different types T, we can exploit name commonality by writing client function templates, like this:

ch11/sum.h

```
template<class T>
T sum(const Array2d<T>& a) {
    T sum = 0.0;
    for (Subscript i = a.shape(0)-1; i >= 0; i--) {
        for (Subscript j = a.shape(1)-1; j >= 0; j--) sum += a(i, j);
    }
    return sum;
}
```

Each class generated from the Array2d<T> class template is itself an interface base, so each function generated from this function template works with instances of any class in the interface category.

Combining name and interface commonality using a class template that generates interface bases is a powerful technique, so we give such a class template a name, a *templatized interface*, and we say that the classes derived from a templatized interface form a *templatized interface category*. As we develop a more complete templatized array interface in Chapter 13, we'll see that it can be useful to cascade this process, deriving templatized interfaces from templatized interfaces.

11.4.3 Nested Typedefs for Name Commonality

In the examples we have shown thus far, the common names supplied by templates and used by client functions have been function or operator names. Type names, supplied by typedef statements, can also be common names supplied by templates and used by clients, as we show in this section.

We could easily write three function templates for computing the sum of the elements in one-, two-, and three-dimensional arrays. These would be declared like this:

ch11/average.C

```
template<class T> T sum(const Array1d<T>&);
template<class T> T sum(const Array2d<T>&);
template<class T> T sum(const Array3d<T>&);
```

We have just seen an implementation of the two-dimensional case. Given these, suppose we then wanted to write a function template for computing the average of the elements in an arbitrary array. All dependence on the array dimensionality is captured in the sum() functions; the only additional information needed for computing the average is the number of elements, available from the numElts()

member specified in the ArrayShape interface. The function template we want to write looks something like this:

ch11/average.C

```
template<class Array>
??? average(const Array& a) {
    ??? s = sum(a);
    return s / a.numElts();
}
```

However, we don't know the type of the array's elements, so we can't complete the code. Parameterizing the function template by both array type and element type like this

ch11/average.C

```
template<class Array, class T>
T average(const Array& a) {    // WRONG code
    T s = sum(a);
    return s / a.numElts();
}
```

will not work: Both template arguments must appear in the function's argument list. (The compiler determines the values of the template parameters from the types of the actual arguments in the function call.) ARM §14.4

Putting a nested typedef in each array interface solves this problem. Here is a modified version of Array2d<T>:

ch11/average.C

```
template<class T>
class Array2d :
    public virtual ArrayShape {
public:
    typedef T EltT;
    // ...
};
```

In all classes generated from the template, the name EltT will refer to the element type. Now we can refer to the element type via the typedef:

ch11/average.C

```
template<class AnArray>
AnArray::EltT average(const AnArray& a) {
    AnArray::EltT s = sum(a);
    return s / a.numElts();
}
```

When this function template is expanded for a particular AnArray, the element type is picked up from the template argument:

ch11/average.C

```
FormedArray2d<double> a(2,3);          // FormedArray2d<T> is in Array2d<T> category
                                       // EltT set to double.
// ... set elements of a ...
double avg = average(a);               // average() returns double
```

Here AnArray is set to Array2d<double>, so the expression AnArray::EltT refers to the name EltT declared in Array2d<double>, which, in turn, is set to double.

Nesting a typedef in a class template allows us to exploit name commonality across a broader range of classes than a particular array interface provides. There is nothing special about the template parameter name AnArray in average(); it does not carry the element type in its name like Array2d<int> or SimpleArray<double> does. The typedef gives a name to the element type that can be referred to via the name of the class, even when we reference it symbolically as in AnArray::EltT. We'll use this technique in Chapters 13 and 15.

11.4.4 Class Templates with Nontype Parameters

The template parameters of a class template need not be types. For example, our rigid arrays use Subscript template arguments to bind the array shape at compile time, as illustrated by the following simple two-dimensional rigid array template:

ch11/rigid.C

```
template<class T, Subscript n0, Subscript n1>
class RigidArray2d :
    public virtual Array2d<T> {
public:
    virtual Dimension dim() const { return 2; }
    virtual Subscript shape(Dimension d) const {
        return (d == 0) ? n0 : n1;
    }
    virtual const T& operator()(Subscript i, Subscript j) const {
        return data[i][j];
    }
    virtual T& operator()(Subscript i, Subscript j) {
        return data[i][j];
    }

    virtual Array2d<T>& operator=(const Array2d<T>&);
    virtual Array2d<T>& operator=(const T&);
private:
    T data[n0][n1];
};
```

When specific Subscript template arguments are supplied as in the code

ch11/rigid.C

```
RigidArray2d<int, 3, 4> a;
```

their values are substituted into the body of the class template to generate a template class. In this example, the constants 3 and 4 specify the dimensions of a built-in array held as a member; they are also used in the implementation of the shape() member function.

Expressions that can be evaluated at compile time can be used to specify template arguments, like this:

ch11/rigid.C

```
const int nrows = 3;
RigidArray2d<int, nrows, 2 * nrows> b;
```

The variable nrows is declared const and initialized by an expression that can be evaluated at compile time. Therefore it can be used as a template argument; likewise the expression 2*nrows can be evaluated at compile time and used as a template argument. This capability allows us to specify that the array should always have twice as many columns as rows. ARM

11.4.5 Templates and Base Class Composition

We have shown how to combine templates with implementation inheritance and how to templatize interfaces. We now consider an example of an interface with a templatized implementation using the base class composition approach of Section 10.7.

The ArrayShape interface base (page 315) specifies a dim() function that returns an array's dimensionality. This function does not itself depend on the array element type T, yet we have left it to be implemented by classes derived from Array1d<T>, Array2d<T>, etc. This arrangement replicates the dim() function for ndim equal 1 for all types T: float, double, complex, GPIBInstrument*, and so on.

We can factor out the implementations of dim into a templatized implementation base using the structure shown in Fig. 11.6. DimensionedArrayShape<ndim>, containing the dim() definition, is derived from ArrayShape, expressing the (partial) implementation of that interface:

ch11/Array2d-with-dim.C

```
template<Dimension ndim>
class DimensionedArrayShape :
    public virtual ArrayShape {
public:
    virtual Dimension dim() const { return ndim; }
};
```

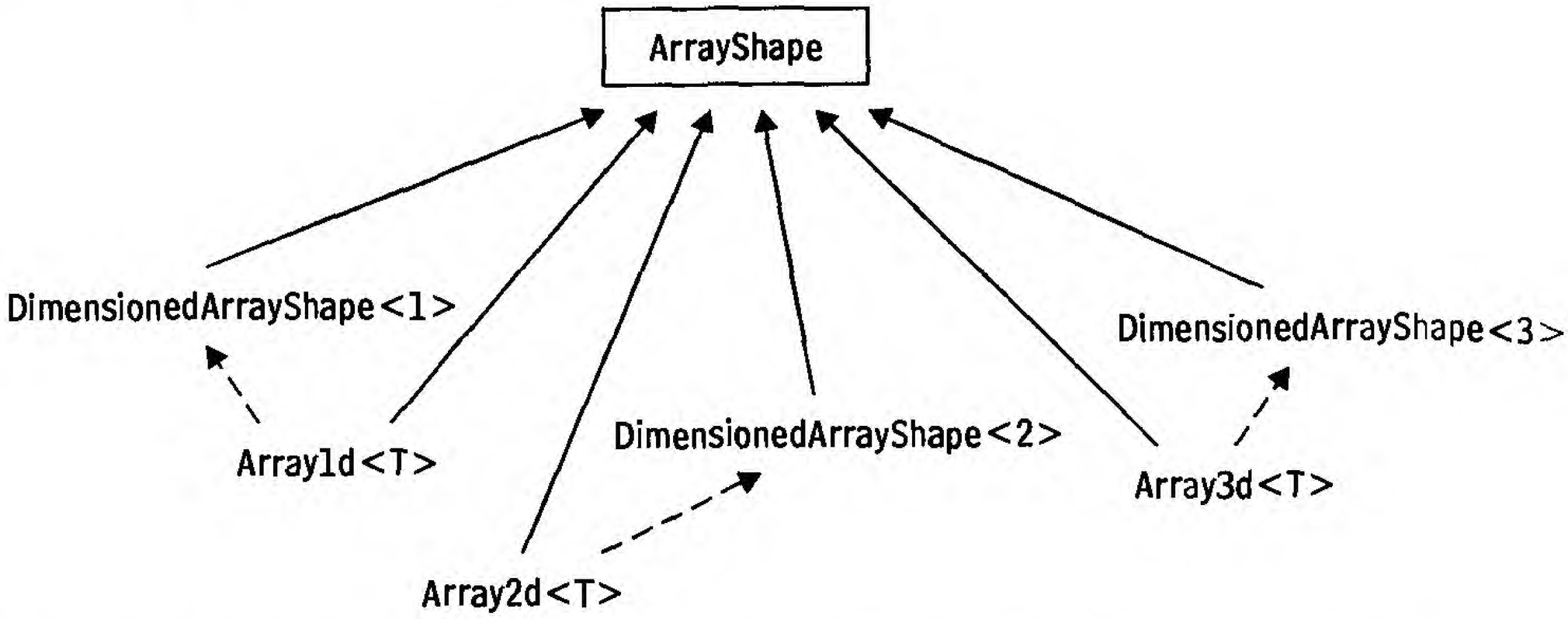


Figure 11.6 DAG Showing Implementation of dim() via Base Class Composition.

Then each of the array interfaces (Array1d<T>, Array2d<T>, etc.) is derived from the appropriate class generated from DimensionedArrayShape<ndim>:

ch11/Array2d-with-dim.C

```
template<class T>
class Array2d :
    public virtual ArrayShape,
    private DimensionedArrayShape<2> {
public:
    // ...
};
```

The dim() function specification left unsatisfied along the Array2d<T> branch of the DAG is satisfied along the DimensionedArrayShape<ndim> branch, and that implementation of dim() will be used for all Array2d<T> objects even when dim() is called through the ArrayShape interface. For Array2d<T>, the dim() in DimensionedArrayShape<2> dominates the (undefined) dim() in ArrayShape.

Admittedly, there is not much to reuse in the implementation of dim(), but the issue is of practical importance in larger examples with similar structure. An alternative that both avoids introducing extra classes and the extra template member function expansions will be used in our version of dim() in Chapter 13. By using inline member functions defined in each specific dimension class, the code space for the template expansions is avoided simply because the function bodies are never compiled out of line. This technique fails for large function bodies that cannot be inlined.

11.5 Global Function Templates

Having templatized all of the structures in Chapters 9 and 10, we return to an important issue introduced when we write global client functions with arguments referencing these classes. As we have seen with examples like sum() and average(), global (nonmember) function templates are common clients of class templates. These functions use name commonality and express implementation structure commonality that extends beyond that of a single template category.

Sometimes function templates are too powerful, generating template functions for argument types you want to handle in a special way. The average() function template on page 321 illustrates this problem. It computes the average of an array's elements, calling a sum() function to accumulate the sum. The sum is accumulated with the precision of the elements themselves.

Suppose you provide a special case for arrays of float elements that accumulates the sum in double precision to reduce numerical error:

ch11/average.C

```
double average(const Array2d<float>& a) {
    double sum = 0;
    for (Subscript i = a.shape(0) - 1; i >= 0; i--) {
        for (Subscript j = a.shape(1) - 1; j >= 0; j--) sum += a(i, j);
    }
    return sum / a.numElts();
}
```

You might then call this function with code like this:

ch11/average.C

```
FormedArray2d<float> b(10, 20);
// ... set elements of b ...
average(b);
```

To your surprise, the double precision version of average() is not used here. This is a consequence of the rules given in Section 5.6 for resolving function calls: Since a function template that matches the argument type FormedArray2d<float> is found, it is used and the ordinary function argument matching process is never applied. This behavior leads us to make the following recommendation:

- Avoid global function templates with argument types that are pure template parameters.

For example, avoid the average() function template on page 321 because its argument type is a pure template parameter and no other average() function can be matched in the presence of this template. On the other hand, the argument of the sum() function template on page 320 is not a pure template parameter, so there is no problem.

In Section 12.7, we will present a general technique that combines inheritance and template expansion to restrict the types for which global function templates are expanded. It allows us to retain control by avoiding pure template parameter arguments while expanding the range of applications for function templates.

11.6 Summary

Classes with member functions or member data differing only in type or other compile-time constants can be implemented as instances of a class template. We can say these classes form a *template category* sharing common names and common implementation structure. Classes in a template category are closely related: They share identical source code. Differences result only when the code is compiled with different template parameters.

As tools for constructing software, templates, inheritance, and interfaces overlap as well as complement one another. The overlap allows us to choose among these tools as appropriate.

Both templates and inheritance support implementation reuse. With templates, source code is reused with different types in place of the template parameter; with inheritance, compiled functions can be reused within the base subobject of derived-class objects. These approaches overlap when a template's parameter is an interface category. In this case, we could rewrite the template as an implementation base containing a pointer or reference to the interface base class as we did in Section 9.8.1. Then different template parameters would become different classes derived from the interface base class.

Both templates and interfaces support common specification reuse. For templates, the common specifications are names used or provided; for interfaces, the common specifications are member functions with argument types fixed by the base class. These approaches overlap when the names used and provided are member functions with fixed argument types. In this case we could declare these member functions to be virtual functions in an interface base class and rewrite template functions as clients of the interface, as we did in Chapter 10. Again different template parameters become different classes derived from a common interface base class.

Described this way, switching between templates and interfaces plus implementation inheritance seems farfetched. The template parameters would have to be just so and the common specification would have to be highly restricted. Most often, which technique to use is clear from the context. But in practice, each technique can be bent to help construct systems more natural using one of the other techniques. To avoid such unnecessary contortions, we showed ways of combining templates with interfaces and inheritance in this chapter. In the next chapter

we again relate these techniques, focusing on their role in building type systems. This will complete the foundation for our exploration of examples in Part III.

11.7 Notes and Comments

11.1 Entire programs can be written without templates: They tend to be programs for problems without structural commonality. Entire programs can be written without interface base classes: They tend to be programs for problems without objects from a wide variety of similar classes. The predominance of linear algebra in scientific computing favors templates; the predominance of topological structures in graphical programming favors interfaces. The availability of both kinds of abstractions broadens the prospects for solving more complex problems.

11.2 Both virtual functions and templates support *polymorphism*, the application of one piece of source code to many kinds of objects. Polymorphism supports abstraction: The details of the many kinds of objects are somehow hidden and only the relevant commonalities of the objects need concern the code that operates on all of them. The polymorphism of virtual functions is dynamic: The type of the object acted on can depend on runtime information. This kind of polymorphism is also called *dynamic binding*, *runtime binding*, or *subtyping inclusion polymorphism* [20]; it is used extensively in Smalltalk. The polymorphism of templates is static: The type of the objects acted on can depend only on compile time information. This kind of polymorphism is also called *generic types* or *parametric polymorphism*. Languages like Ada support this kind of polymorphism, and Axiom [61] demonstrates that it can be supported at runtime. Ultimately, polymorphic collections like the one discussed in Section 9.9 are the only important structure that cannot be programmed with generic polymorphism alone.

In this chapter's summary we sketched out the parallels between dynamic and parametric polymorphism but avoided these terms directly in favor of the C++ specific ones, virtual function interfaces and templates. We did this because the term polymorphism has several subtly different meanings. Polymorphism describes type relations. Various programming languages support various kinds of relations, various times when the relations can be built, and they vary on how and when the relations are checked.

11.8 Exercises

11.1 Define an overloaded tuplize() template (page 305) to allow output of CheckedSimpleArray<T> (page 309). Then redefine this template to allow it to work for ArrayId<T>.

11.2 Reimplement Fallible<T> (page 308) to use a pointer object member datum that is tested against zero for validity.

11.3 Create a two-dimensional floating point array by using CheckedSimpleArray<T> with T equal to CheckedSimpleArray<double>. Illustrate its use and abuse; discuss its disadvantages.

11.4 Use SimpleArray<Subscript> to implement a SimpleArrayShape class derived from ArrayShape. Use SimpleArray<T> to implement SimpleArray1d<T> derived from your SimpleArrayShape and from the Array1d<T> interface.

11.5 Apply sum() (page 319) and average (page 321) to the arrays from Exercises 11.3 and 11.4. Fix the array classes if they do not work the first time.

11.6 Refactor the code from Exercise 11.4 using the base class compositions from Section 11.4.5.

11.7 Design classes that can have sum() and average() members to replace the global template functions from page 319 and page 321. Compare the range of array classes that can be summed and averaged by your classes compared to the global template functions.

CHAPTER 12

Types

Interfaces, inheritance, and templates provide tools for expressing commonality among classes and functions. We have looked at how to use these tools to express various kinds of commonality, but we have glossed over their connection to each other: They all depend on the ability to define new types and relationships among types. In this chapter, we dig deeper to understand the connection between these tools and types. Part of this material will be review and reanalysis of examples from preceding chapters; part will be new techniques for working with types.

Every C++ object has a *type*, a specification of both the content of the object and what can be done with the object. For objects of built-in type, both aspects are specified by C++; for class objects, the object's class definition specifies both. Although every object has a type, not all types are used to create objects. Abstract base classes specify what we can do with objects that are instances of classes derived from the abstract base, but they do not specify what such objects contain.

In this chapter we take the view that a type specifies rules for manipulating objects. Starting with the concept that different rules govern computing with int and with double objects, we move on to see that classes allow the programmer to set the rules, that the rules in one class can be related to the rules in another by the declaration of common public base classes, and that rules can be built up by inheritance of the rules in existing classes. Class templates can then be viewed as "rule functions." These layers of increasing sophistication can be daunting, but ultimately they are a key aspect of what makes C++ suitable for tackling sophisticated problems.

Throughout our discussion we face the double-edged sword of rules: They provide both essential order and inevitable restriction. Types tell us how objects can be used properly and how they may not be used. As we work to define new types and to define more complex relations among types, the rules in a type can conflict with each other, or rules in one type can conflict with those in another, or we can simply end up with systems of rules that don't act like we planned. These problems appear as compile errors: The compiler is using the type information you provide to help you build a consistent system. In our experience, reworking

types to build a consistent system that can be compiled is a big step toward getting a program working, much more so than in languages like FORTRAN and C.

12.1 Basic Ideas of Type

The most basic type rules are those specified by C++ for each built-in type. The rules for the built-in types specify their structure, how objects of one type can be created from objects of another type, and what operations each type supports. For example, C++ specifies the relative sizes of float and double, allows a float object to be created from an int object, and provides addition for int and for float but does not allow pointer dereferencing on double objects.

Each built-in type also has several other types associated with it: pointers, references, and arrays. The rules for int* and int& and int arrays are independent of, but given in terms of, the rules for int. Thus the dereferencing operator for pointers, *, always gives a reference to an object of the corresponding type: for int* p, *p is an int, whereas for double* q, *q is a double.

The type rules for pointers are independent of the particular object type, so we can discuss them in terms of a parameter T for the actual type. For example, given the declaration T* r, *r will give an object of type T. For this reason, pointers, references, and arrays have a *parametric type* relationship with their corresponding object type. The T in T* r acts like a parameter that can take type values: int, float and so on. Since C++ has templates as programmer-defined parametric types (Chapter 11), we call pointers, references, and arrays *built-in parametric* types. (See Notes and Comments 12.1.)

All of the built-in types and built-in parametric types specify their objects' behavior and state structure. Type relationships also exist, such as the usual arithmetic conversion rules and the parametric relation of int* to int, but the relationships are fixed and defined by C++. This is the most basic view of the idea of type: a complete specification of state and behavior as given by the programming language. This is the end of the story for types in languages like FORTRAN and C.

12.1.1 Classes as Programmer-Defined Types

In C++ a completely new type is introduced by each class definition. This facility to extend the fixed set of built-in types alters the programming process dramatically. Rather than decomposing a problem into functions that operate on built-in type objects, you design new types adapted to the problem. The advantage accumulates in large and complex programs solving problems unlike those faced by the designers of the built-in types.

A class name is synonymous with the type specified by its definition: Class names identify types and therefore must be unique. For example, we can create a new type called Matrix like this:

ch12/tMatrix.C

```
class Matrix {
    public:
    // member functions, etc.
};
```

If we duplicate the preceding class definition, replacing Matrix with YazooMatrix, we would create a new and different type not connected to the Matrix class in any way that C++ can detect.

If we define a completely different class in the same program and give it the name Matrix, the C++ compiler will flag the second one as an error: Two types cannot exist with the same name. Beyond the C++ requirement that each type name be unique, human readers need to be able to imagine what each class represents:

- Select meaningful class names.

For example, rather than a generic name like Matrix, pick a name that describes the character of the type, such as RowMajorMatrix2d. The extra effort required to type a long class name is repaid as a program grows more complicated. If, within a local context, long names become difficult to read or otherwise distracting, typedef statements can be used to provide local abbreviations.

A class definition also defines built-in parametric types implicitly—pointers, arrays, and references—that act for a class type T just like they would for a built-in type T. For example, we can write

ch12/tMatrix.C

```
Matrix m;           // We can make objects if Matrix is a class,
Matrix* mp;         // ... and pointers,
Matrix& mr = m;     // ... and references,
Matrix ma[4];       // ... and arrays.
```

C++ also provides facilities for creating new parametric types—template classes. Some of these are directly analogous to the built-in parametric types, as illustrated by the array classes we have discussed in Chapter 11 and the pointer classes we will introduce in Chapter 14. Others may introduce type relations in completely new ways unrelated to the built-in type system, as we do with the function-structure category construct we will introduce in Section 12.7.

12.1.2 const Types

Using const type modifiers helps in writing robust code. Adding a const modifier to the type of an object changes the type it will be created with. Consider the following code:

ch12/tTypeErrsEg.C

```
const SimpleArray<int> a(5);
int n = a.numElts();            // ok
a[0] = 1;                       // Compile error!
```

Obviously the type of a is not exactly the type of the class SimpleArray<int>. Only part of the specification in that class applies to a: the object layout, the constructor, and the const member functions like numElts(). Assignment to elements, valid for SimpleArray<int> objects, is not valid for a.

The type of a is const SimpleArray<int>. Objects of this type work only with those member functions in the type SimpleArray<int> that are declared const. Every built-in and class type in C++ has a related const type.

We can view const types as built-in parametric types somewhat similar to pointers, references, and arrays: For every type T, there is a const T type formed by the const member functions of the class. We can also view const types as restricted types: The specification of the const T type is the subset of the specification of type T that is declared const.

C++ also defines built-in rules for conversions involving const types. A const object cannot be used to initialize a non-const reference, so

ch12/tTypeErrsEg.C

```
const int x = 3;
int& xr = x;    // WRONG: const int cannot be converted to int&
```

is illegal. If this were allowed, a const object could be modified simply by using it to initialize a non-const reference (or pointer).

On the other hand, a non-const object can be used to initialize a const reference:

ch12/tTypeErrsEg.C

```
int y = 4;
const int& yr = y;    // OK
```

This simply means that although y can be modified, when accessed via yr it can't.

Using an object of type T to initialize a restricted type of const T is an example of a type relationship in C++. As we will soon see, C++ uses the same concept to express is-usable-as commonality: A base class reference can be initialized with a derived class object.

12.1.3 Object and Variable Types

Our discussion of built-in parametric types touches on type relationships, the core subject of the next section. For example, every class type is related to a const type, every type has a related pointer type, and, moreover, there are const pointers and pointers to const types.

Before moving on, recall our discussion of Section 2.2 or Section 3.2. There we said that both objects and variables have types. As we encounter more sophisticated ideas of type, the association of type with both objects and variables becomes more important to understand. When a client function operates on an argument variable the function knows the variable's type; when the client function

is called the caller knows the object type it will supply as an argument. If the connection between variable type and object type is flexible, then the client function can be reused on a wider range of objects.

Two concepts—indirection and type relationships—are used by C++ to allow variables of one type to work on objects of a related type. Let's review these concepts with simple examples before tackling interface types.

Indirection means that reference and pointer variables can refer to an object just like the variable directly bound to the object. Moreover, expressions that evaluate to temporary references or pointers can refer to objects. For example, here both a and r refer to the same SimpleArray1d<int> object:

ch12/tMatrix.C

```
SimpleArray<int> a(5);
SimpleArray<int>& r = a;
```

Certainly both a and r have type: We know how to use them in expressions because of their type. We should also view the object that they both refer to as having a type. The bits in storage are arranged according to the rules of SimpleArray<int> and they can only be transformed correctly according to those rules. Indirection allows us to have multiple views of an object by having multiple variables reference it; each view and the object itself has a type.

Type relationships allow a reference or pointer to an object to have a type other than the object's type. Thus we can create a const reference to our array object:

ch12/tMatrix.C

```
const SimpleArray<int>& cr = a;
```

The object a has not changed: It is still a SimpleArray1d<int> object. However, the type of the variable cr prevents those function calls through cr that modify a. The object and the reference variable have distinct types, related through C++'s built-in const type relationship.

Indirection and type relationships together create multiple views of objects. The association of a const reference variable with an object creates a view of an object through a different type. For example, in the foregoing code, cr is a const view of the object a. Actions on the object through the reference are limited to the const member functions.

Type relationships are sometimes called type conversion rules. Don't confuse association through indirection with conversion of object values. Conversion creates a new object of the new type, not a new association with a new type that references the original object.

So far we have only discussed the built-in type relationship between const and non-const types. The next section discusses interface base classes as a way of building user-defined type relationships. The C++ type system exploits these

relationships to achieve simultaneously the type checking necessary for writing correct programs and the generality necessary for writing sophisticated programs.

12.2 Types and Interfaces

The preceding section laid the groundwork for reexploring interface categories from Chapter 9, looking in detail at their type implications. We know that classes are types, that objects have types, and that const T& reference variables can refer to objects of type T. Now we look at interfaces as programmer-defined rules for creating reference variable types for alternative views of objects.

12.2.1 Creation and Use Types

A class written to allow objects to be created must specify completely both the structure of the objects' state and behavior. All of the classes in Part I were written in this way, as were the classes in Chapter 9 that represented specific VoltageSupply objects, like Acme130_VS (page 235).

In Chapter 9 we introduced a different kind of class, an interface base class, that does not exist for creating objects. Interface base classes, like the abstract interface VoltageSupply (page 234), do not have member data and generally contain pure virtual functions and no constructors. No VoltageSupply objects can be created. But the VoltageSupply specification can be declared for a class like Acme130 or VoltOn59, allowing VoltageSupply pointers and references to be created that access these objects indirectly.

In the following code, there are two objects, one with type Acme130_VS and one with type VoltOn59_VS, and two references, one for each object. The two references have the same type; the two objects have distinct types:

ch12/tRefsNObjs.C

```
Acme130_VS a(gpib, 12);
VoltageSupply& ar = a;
VoltOn59_VS v(gpib, 13);
VoltageSupply& vr = v;
```

Initializing VoltageSupply references to both Acme130_VS and VoltOn59_VS objects is legal because they are publicly derived from the VoltageSupply base class.

A similar example is shown on page 235: two objects of two distinct types were created and two VoltageSupply references were initialized to the objects. In that example the references were function arguments and we did not discuss the issue of type; here the references are explicit to highlight the role of a common base type reference to two different types of objects.

Publicly deriving Acme130_VS from the interface base class VoltageSupply creates a new type relationship: References of type VoltageSupply can be initialized to

refer to objects of type Acme130_VS. This relationship is not parametric and is completely under the programmer's control. Functions can be written for VoltageSupply references or pointers without any knowledge of Acme130_VS, and only by public derivation in Acme130_VS do we establish the relationship.

A class, like Acme130_VS, that specifies state and behavior completely can be called a *creation type* or an *implementation*. Interface base classes like VoltageSupply that specify behavior only partially can be called *interface types*; they are a special case of a *base type*, the type of a public base class for a creation type.

Every object is an instance of one and only one class and hence has one and only one creation type. Although we might think of, say, the "VoltageSupply" types, any particular VoltageSupply object must be either an Acme130_VS or a VoltOn59_VS or some other creation type.

References or pointers, however, can be created for any type. Client functions are written with arguments whose type can be references to creation types, interface types, or other base types. We call an argument type a *use type*. Public derivation of creation types from the base types allows a use type to be initialized by objects with the creation type. Public derivation is the C++ specification that allows this type relationship. The phrase "using an object through an interface" means that a reference or pointer in a client function is bound to an object of a creation type that is derived publicly from the interface type.

12.2.2 Type Checking with Interface Types

Ultimately types are important because they allow type checking; create/use type relationships are important because they allow type checking of use independent of creation. Once we create an object, the type it has is fixed and definite, but we can still use an object as an object of another related type while preserving type checking at compile time. This separation allows new creation types to be added and then used in existing code without changing the existing code.

Creating an object requires knowing its creation type—its class—so that memory can be allocated to hold its state and the state can be initialized. For example, when an array is created, its dimensionality, shape, element type, and data layout must be known. However, using an object requires only being able to call some of its member functions. These member functions can be listed in an interface base class, giving a *use type* for client function type checking.

Let's see how this works. Suppose that you want to write a function that computes the sum of the squares of a one-dimensional array with float elements, regardless of whether the array is rigid, formed, or elastic. Conceptually this only requires knowing that the array is one dimensional, that its shape can be found, that its elements can be accessed, and that these elements have type float. The interface category Array1d<float> (page 319) specifies the common behavior of all one-dimensional arrays with float elements; Array1d<float> is a use type.

To compute the sum for any array in the `Array1d<T>` category, declare the argument to be a reference of the interface type. Then the function accepts as an argument any array in the category:

ch12/sumsq.C

```
double sumsq(const Array1d<float>& a) {
    double sum = 0.0;
    for (int i = a.shape(0)-1; i >= 0; i--) sum += a(i)*a(i);
    return sum;
}
```

The sumsq() function can be called with any member of the category as an argument, as illustrated by the following examples:

ch12/sumsq.C

```
// Read a size from stdin, create a 1-d array of that size,
// read the array element values, and compute the sum of squares.
int n;
cin >> n;
FormedArray1d<float> fa(n);
cin >> fa;
cout << "Sum of the squares of " << fa << " is " << sumsq(fa) << endl;

// Read a 5-element array from stdin and compute its sum of squares
RigidArray1d<float, 5> ra;
cin >> ra;
cout << "Sum of the squares of " << ra << " is " << sumsq(ra) << endl;
```

Although applying sumsq() to both `FormedArray1d<float>` and `RigidArray1d<float,5>` instances seems natural and sensible to us, recall that C++ must be able to check the type of function arguments against the actual type of the instances supplied in the function call. C++ must check that only `Array1d<float>` operations are called in sumsq() while allowing both `FormedArray1d<float>` and `RigidArray1d<float,5>` objects as arguments.

C++ achieves this important combination of security and flexibility by requiring objects to be created with the fixed, definite type of their class but allowing these objects to be used as instances of any base class. Both `FormedArray1d<float>` and `RigidArray1d<float,5>` are derived publicly from `Array1d<float>`, and objects in those classes can be used as `Array1d<float>` objects.

The C++ mechanism that allows sumsq() to apply to both `FormedArray1d<float>` and `RigidArray1d<float,5>` objects has two components. First, these classes are derived from the common base class, `Array1d<float>`, which forms a *base type*. This derivation ensures that all `FormedArray1d<float>` and `RigidArray1d<float,5>` objects conform to the specifications for the `Array1d<float>` type. Second, C++ allows

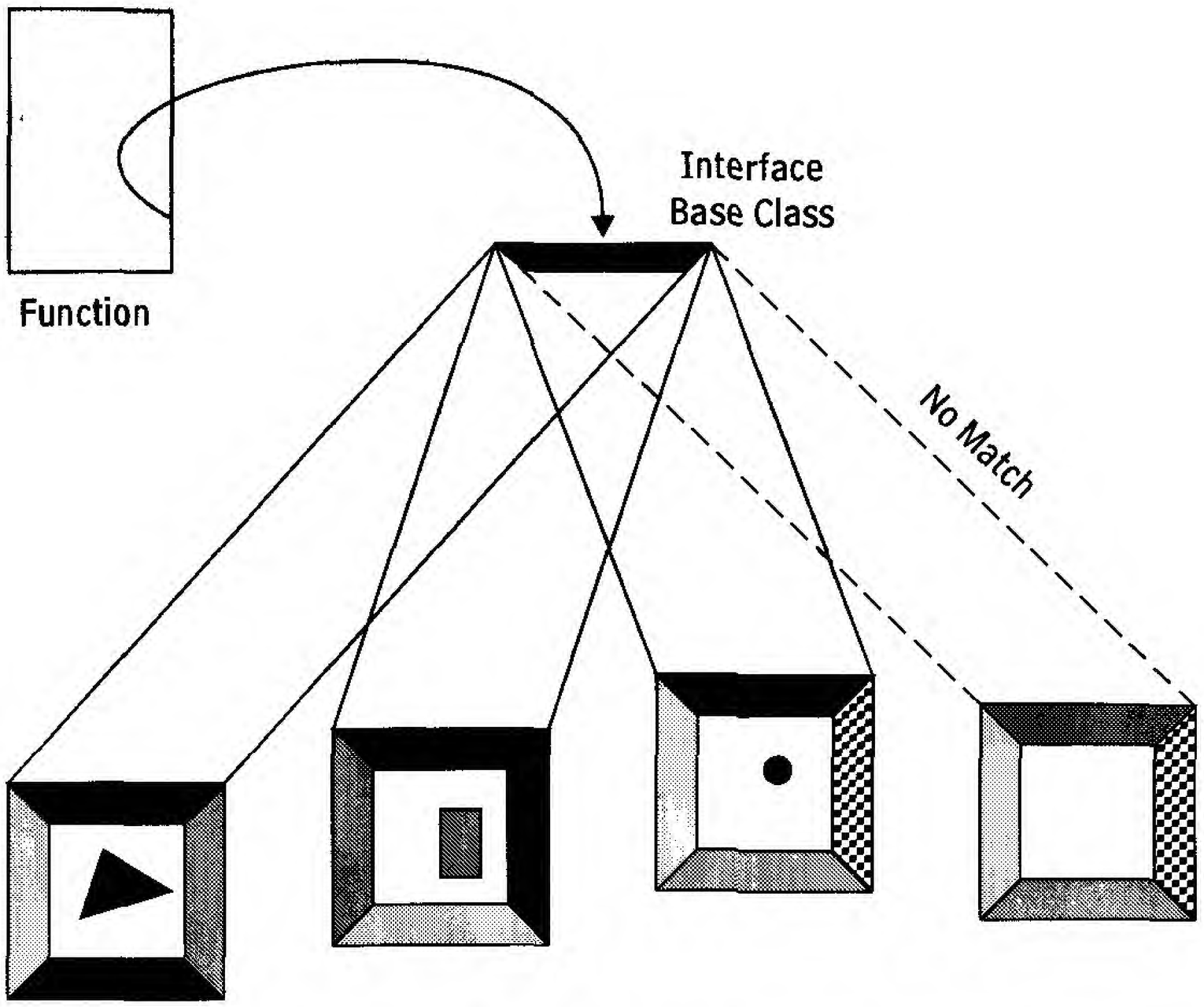


Figure 12.1 The Function on the Left Can Manipulate an Object Using Only Functions in an Interface Base Class. We sketch this by showing an arrow meaning"uses" that points to a sketch of the interface. Any object with that interface can be manipulated safely, but any object without that interface cannot be used. We indicate this by showing objects of different types (boxes with triangle, rectangle and circle inside) matching the interface (same gray shaded shape used by the function). Note that the interface base class is a class, not an object; therefore it does not exist while the program runs.

references to the `Array1d<float>` interface type to be initialized with `FormedArray1d<float>` or `RigidArray1d<float,5>` objects or objects from any other class derived from `Array1d<float>`. This allows objects of both derived types to be used in code written using references to their interface types. Derivation relates the types of `FormedArray1d<float>` and `RigidArray1d<float,5>` objects to the interface type `Array1d<float>`; initialization of the `Array1d<float>` reference with either `FormedArray1d<float>` or `RigidArray1d<float,5>` objects allows instances of these specific types to be used as objects of the interface type.

Figure 12.1 illustrates this idea of type checking with interfaces. To use C++ effectively, we seek to

- Define an object using the type of its class and use it via the type of its class or any of its interface types.

Note that there are no instances of `ArrayId<float>`; it is a use type. But there are objects that behave according to the specification given by `ArrayId<float>`: the instances of any class in the category `ArrayId<float>`. Accessing objects through references to any of the object's interface types is a central concept in the C++ type system.

A class can be derived from more than one base class, and a base class can be derived from another base class: `ArrayShape` is an interface type for `ArrayId<T>`, which in turn is an interface type for `FormedArrayId<T>`, and so forth. Consequently an object may have many interface types, some of which also have their own interface types.

Everything said here about references to objects applies equally well to pointers to objects. Specifically, an interface type pointer can be initialized by a pointer to an object in any class derived from that interface type.

Effective use of this powerful language facility requires persistent and thorough abstraction of common features of classes into interface categories and disciplined use of only these common features in code written for classes in the interface category. C++ compilers can type check the use of interface type references and pointers and their initialization, but they cannot isolate the common functions in a group of classes and create interface types automatically. You must do that.

12.3 Type Conversions

Among the relations between types, conversions can be the most troublesome and powerful. Whereas parametric type relations like `T*` and `T` give us clear ways to work with objects, conversions build in relations that can alter the way we work with objects. We touch on some of the type-related issues of conversion in this section.

12.3.1 Built-in Conversions

We have already discussed various built-in conversions on page 23. The important new conversion we need to explore now is conversion of base class and derived-class pointers and references.

The fundamental mechanism of object-oriented programming in C++ uses conversion of a derived-class reference to a base class reference. Specifically, a base class reference can be initialized with a derived-class reference or object automatically. The object itself is not converted or in any other way altered. The allowed initialization is a specific type relationship built into the language: Base class reference types are related to derived-class types by allowed initialization. The opposite initialization is not allowed (see Section 12.4).

12.3.2 Programmer-Defined Conversions

In Section 6.4.2, we described conversion operators that allow objects of one class to be created from objects of another. Conversion operators would be more accurately labeled *construction operators* because they do not alter the type of the object they apply to and most often they don't alter the object's state. Instead conversion operators are the logical inverse of constructors: Rather than specifying how an object of the type can be constructed from given arguments, they specify how an object of a given type can be constructed from the object to which the conversion operator is applied.

We can view conversion operators as a programmer-defined type relationship paralleling the type relationships that C++ provides. For example, C++ provides conversion from float to double:

ch12/conversion.C

```
float f = 3.1;
double d = f;
```

Here f is unaltered, but d's value results from a conversion from float to double.

We get the same kind of behavior with programmer-defined types whenever a conversion is defined by a conversion operator or a single argument constructor. For example, the commonly available complex class is based on double precision floating point. We can create a single precision version that supplies a conversion operator to the existing type:

ch12/conversion.C

```
typedef complex ComplexDouble;

class ComplexFloat {
public:
    ComplexFloat(float, float);
    operator ComplexDouble();
    // ...
};
```

Then we can write

ch12/conversion.C

```
ComplexFloat f(1.0, 2.0);
ComplexDouble dr = f;
```

and cause the initialization of a new complex number directly from our ComplexFloat object.

The type system also plays a role in determining when C++ creates temporary objects. The following code looks superficially similar to the binding of an interface base class reference to an object:

ch12/conversion.C

```
float f = 3.1;
const double& dr = f;
```

However, there is no double interface for float objects. The reference variable dr is initialized to refer to a temporary double object created by conversion from the float object f.

Notice that we declared the double reference to be a const reference. This is required, and the code will not compile without the const specification. Here's why. Since a reference to a double is needed, the compiler arranges to create an unnamed double object initialized from the float object f. Such a *temporary object* is const: C++ assumes that altering an object that is not associated with a variable is a mistake. Since dr is initialized to refer to the temporary, it must be declared const. Thus the foregoing code is equivalent to the following code:

ch12/conversion.C

```
float f = 3.1;
const double temp = f;
const double& d = temp;
```

12.4 Loss of Type Information

Fundamentally, abstraction is the process of ignoring inconsequential differences and concentrating on important similarities. To use Acme130_VS and VoltOn59_VS objects as VoltageSupply objects, we choose deliberately to ignore any special features of the particular supplies. As a result, initializing a VoltageSupply& with an Acme130_VS object discards information: A function written for a VoltageSupply does not know what kind of supply it is using.

In many cases, this information loss is not important. To manipulate an Acme130_VS as a VoltageSupply with some special added features, we write as much code using functions that call through the VoltageSupply interface as seems reasonable. Then we write the remaining function directly for Acme130_VS type objects.

In some cases, the information loss is important, but recovering the information is simple. The function-structure category classes that we introduce in Section 12.7 illustrate such a situation.

There are other cases in which the loss of type information is important and recovery more difficult. This happens, for example, when we build a collection of unrelated objects using base class pointers. Consider a program that uses GPIB components to run an instrument. The program may need to list the components currently allocated for an experiment, including not just the GPIB instruments but also the cables and controller computer boards. Voltage supplies, cables, and controller boards do not share "is-usable-as" commonality; they share "are-needed-together" commonality. They are all items needed for a particular project, but they are not used for that project in a common way.

Let's proceed by assuming that we have a class (e.g., GPIBComponent) that represents the "are-needed-together" commonality of all the GPIB instruments. We then publicly derive each class that represents a kind of GPIB component (Acme130, etc.) from GPIBComponent to give the classes the "are-needed-together"

commonality. The list of components allocated for an experiment could then be a list of pointers to this base class, (e.g., a List<GPIBComponent*>). (The List<T> class template appears on page 170). All objects that are instances of a class derived from GPIBComponent can be placed in the collection by storing the component's address in the list. Conversion of the pointer to the object to a GPIBComponent* is allowed by the public derivation.

We can understand the information loss problem without knowing the content of the GPIBComponent class. Adding the GPIBComponent base class to our Acme 130 class from page 244 gives a new version we call Acme130_GC. Similarly, we can define GPIBController_GC_GC as a version of GPIBController_GC (page 247) created by deriving from GPIBComponent (see Exercise 12.4). This done, we can add a pointer to an Acme130_GC or a GPIBController_GC_GC to a list of GPIBComponent*s, as in the following function:

ch12/tGPIBComponent.C

```
void addYourComponents(List<GPIBComponent*>& a) {
    GPIBController_GC_GC* pc = new GPIBController_GC_GC();
    a.add(pc);                                  // Public derivation: ok to convert
    a.add(new Acme130_GC(*pc, 12));             // Public derivation: ok to convert
}
```

Type information is lost when this function is run. Only by reading this function can we know that the list of GPIBComponent pointers has a pointer to an Acme130_GC and a pointer to a GPIBController_GC_GC. Even then we don't know which list element points to which kind of component unless we have some way to test at runtime.

Compare this list of GBIBComponent pointers to the array of GPIBInstrument pointers in Section 9.9. There we isolated common operations in GPIBInstrument and passed the operations along to each element of the collection. For GPIBComponents, however, there are no common operations: The commonality being captured is "are needed together." To use the objects on the list, we must somehow recover the information about the objects' types.

12.4.1 Type Recovery with Type Codes

One solution is to have each component be able to return a type code represented as an enum in the GPIBComponent base class, as shown in our first version of GPIBComponent, called GPIBComponent_TC with TC meaning "type code":

ch12/GPIBComponent_TC.h

```
class GPIBComponent_TC {
public:
    enum Kind { Acme130Component, GPIBControllerComponent /* ... */ };
    virtual Kind kind() const = 0;
    virtual ~GPIBComponent_TC() {}
};
```

Each class derived from GPIBComponent_TC must override kind() to return the appropriate value, as in this version of Acme130:

ch12/Acme130_GCTC

```
class Acme130_GCTC :
    public virtual VoltageSupply,
    public virtual GPIBInstrument,
    public         GPIBComponent_TC {
public:
    virtual Kind kind() const { return Acme130Component; }
    // ... Rest like Acme130_VS_GI_GC...
};
```

The specification of the kind() function is the same in each derived class, but each GPIBComponent_TC object returns a value specific to its derived type. (The derivation from GPIBComponent_TC cannot be virtual for a technical reason explained at the end of this section.)

When operating on a list of GPIB components, we call kind() to determine which kind of object we really have, like this:

ch12/tGPIBComponentTC.C

```
List<GPIBComponent_TC*> components;
addYourComponents(components);
// At this point, the component types are lost

for (ListIterator<GPIBComponent_TC*> it(components); it.more(); it.advance()) {
    GPIBComponent_TC* component = it.current();
    GPIBComponent_TC::Kind kind = component->kind();
    if (kind == GPIBComponent_TC::Acme130Component) {
        Acme130_GCTC* p = (Acme130_GCTC*) component;
        cout << "An Acme 130 voltage supply; range: "
             << p->minimum() << "-" << p->maximum() << endl;
    }
    else if (kind == GPIBComponent_TC::GPIBControllerComponent) {
        GPIBController_GC_GCTC* p = (GPIBController_GC_GCTC*) component;
        cout << "A GPIB Controller" << endl;
    }
}
```

The expression (Acme130_GCTC*) component is a *type cast*, or simply a *cast*. Syntactically a cast consists of a type enclosed in parentheses preceding an expression. Its effect is to treat the value of the expression as if it were of the specified type. In this example, the cast says to treat component as if it were an Acme130_GCTC* instead of a GPIBComponent_TC*. As with conversions, the value of component is not altered,

but p is initialized from the result of treating component as an Acme130_GCTC*. We are saying that we know something the compiler doesn't: We know, by means not apparent to the compiler, that component really points at an Acme130_GCTC object.

The particular form of the type casts in this code is called a *downcast* because pointers are being converted "down" the class DAG—from base type to derived type—instead of "up" the DAG—from derived type to base type. Conversions from derived type to base type discard information, but conversions from base type to derived type add information, going from one type to a more specific type. We know this extra information only by the logic of our code, which the compiler can't check.

Therefore casts are dangerous. If you make a mistake, the compiler can't detect it. As your program evolves, you must ensure that all the casts remain valid, implying that you must find all the casts that could be affected by your changes.

- Avoid type casts.

Even if we type cast correctly, type-testing code is a software maintenance problem. As we derive new types from GPIBComponent_TC, the compiler can't warn us to add new type cases to client functions. Also, C++ prohibits downcasts from a virtual base, forcing us not to use virtual derivation from GPIBComponent_TC and thus preventing use of the base class composition style we described in Section 10.7. Type testing and type casts are the most straightforward way to solve the type information loss problem we have described, but such code and its lurking dangers can be avoided.

AF

12.4.2 Type Recovery with Double Dispatch

Our previous solution to the type information loss problem was to define an interface that required each GPIB component class to provide a virtual function identifying its creation type. We then used the information supplied by that function to do an appropriate type cast so that we could perform some action on each component.

The solution we present in this section turns the problem around, taking a more object-oriented view: Instead of figuring out what kind of object we are dealing with and then doing something appropriate to that kind of object, we specify the desired action for each kind of object and say "do it."

As in Section 12.4.1, we define an interface base class for all GPIB components and again use a virtual function to recover the lost type information. This time, however, the virtual function doesn't tell us what kind of object we are dealing with. Instead we give the virtual function an argument that specifies the actions we want done on each kind of GPIB component object. The specification of this function—dispatchTo()—in GPIBComponent is derived-type independent:

ch12/GPIBComponent.h

```
class GPIBComponentAcceptor;      // See below.

class GPIBComponent {             // Base class for lists of GPIB components
public:
    virtual void dispatchTo(GPIBComponentAcceptor&) const = 0;
    virtual ~GPIBComponent() {}
};
```

The argument for the dispatchTo() function is a class in the GPIBComponentAcceptor interface category, as specified by the following interface base class:

ch12/Acme130_GC.h

```
class GPIBComponentAcceptor {
public:
    virtual void acceptAcme130(const Acme130_GC&) = 0;
    virtual void acceptGPIBController(const GPIBController_GC_GC&) = 0;
    // ...
    virtual ~GPIBComponentAcceptor() {}
};
```

Each *accept function* specifies an action for the corresponding type and is passed a reference to the kind of object for which it provides the action. To supply an action, derive a class from GPIBComponentAcceptor and override the appropriate accept function.

Each kind of GPIB component provides an implementation of dispatchTo() that calls the appropriate accept function, passing a reference to itself. Here is the implementation for Acme130_GC:

ch12/Acme130_GC.h

```
class Acme130_GC :
    public virtual VoltageSupply,
    public virtual GPIBInstrument,
    public virtual GPIBComponent {
public:
    virtual void dispatchTo(GPIBComponentAcceptor& a) const { a.acceptAcme130(*this); }
    // ... Rest like Acme130_VS_GI_GC...
};
```

We can call dispatchTo() on any GPIBComponent object. If the component happens to be an Acme130_GC, the GPIBComponentAcceptor's acceptAcme130() member function will be called. Thus a virtual function call with a type-independent argument moves us into a member function specific to an object's creation type and it calls one of the creation-type-specific functions listed in GPIBComponentAcceptor.

Objects in the GPIBComponentAcceptor interface category accept GPIB components of each possible type and take actions on them. The result depends on the

action coded in the acceptAcme130_GC() function for the particular GPIBComponent-Acceptor dispatched to. For example, we might just print out a message for each kind of component:

ch12/tGPIBComponent.C

```
class GPIBComponentPrinter :
    public GPIBComponentAcceptor {
public:
    virtual void acceptAcme130(const Acme130_GC& p) {
        cout << "An Acme 130 voltage supply; range: "
             << p.minimum() << "–" << p.maximum() << endl;
    }
    virtual void acceptGPIBController(const GPIBController_GC_GC& p) {
        cout << "A GPIB Controller" << endl;
    }
    // ...
};
```

This class acts like a package of operations, one for each type of object we can encounter on any list of GPIBComponents.

When we want to operate on a list of GPIB components, we pass this package of operations over the list:

ch12/tGPIBComponent.C

```
List<GPIBComponent*> components;
addYourComponents(components);
// At this point, the component types are lost

GPIBComponentPrinter printer;
for (ListIterator<GPIBComponent*> it(components); it.more(); it.advance()) {
    it.current()->dispatchTo(printer);
}
```

Each call to dispatchTo() potentially results in a call to a different part of our package of operations.

For the GPIBComponentPrinter implementation of the GPIBComponentAcceptor interface, this results in the output:

```
(Acme130_GC now at address 12)
A GPIB Controller
An Acme 130 voltage supply; range: 0–10
```

Two virtual function calls are made for each operation, one to invoke the dispatchTo() function in the GPIB component object and one to invoke the appropriate accept function. Virtual function calls are sometimes referred to as a *dispatch* to an object, so this technique has come to be called *double dispatch*.

For simplicity, we used printing as the action in this example. We could have added a print() virtual function to the GPIBComponent interface instead. However, suppose we want to send a command to all controllers on the list without affecting other kinds of components. We could add a sendOnlyToControllers() member function to the GPIBComponent interface, but this would require us to add a function to every GPIB component class, not just the controller class. As our program grows, we would end up with a *fat interface*, an interface base class with so many functions that type checking against it will not find many errors of usage.

Double dispatch allows us to leave the classes alone and add new function by creating a new class derived from GPIBComponentAcceptor. Double dispatch's cost comes each time we wish to add a new kind of object to our list: We must add a new function in the GPIBComponentAcceptor. Additions cause us to add new functions to each GPIBComponentAcceptor implementation. These new functions represent the work we must do to support the new type in that implementation. Thus the requirement that we add new functions is exactly how type checking aids in building consistent programs.

12.5 Types and Class Templates

Having looked at the type implications of interfaces, we now discuss the type implications of class templates. A class template, like SimpleArray<T> (page 101), looks like a type, but it is not. There are no objects of type SimpleArray<T>, nor can there be pointers or references to objects of type SimpleArray<T>. Yet we often write code in which SimpleArray<T> is used just like a type in variable declarations, argument declarations, and return type specifications. Our challenge here is to understand what a class template means with respect to the C++ type system.

12.5.1 Roles of a Class Template

The class template SimpleArray<T> serves two purposes. First, it specifies that names of the form SimpleArray<T>, with T replaced by a type name, are names of classes. Such a name is called a *template class name*. SimpleArray<int> and SimpleArray<float> are template class names. As names of classes, they are also names of types. Second, the class template provides a specification for generating a family of template classes all having the same structure. Each generated template class is a type: SimpleArray<int> and SimpleArray<float> are types.

Usually a class template serves both purposes simultaneously: We use a template class name, and the compiler generates for us the corresponding class. Sometimes, however, we may want to have a family of classes with template class names but with different implementations. For example, when in Chapter 13 we implement the family of array classes described in Section 11.3, we'll need a simple, low-overhead way to store arrays of subscripts. We'll use a SubscriptAr-

ray<ndim> class template, with the parameter being a Dimension that specifies the number of subscripts:

Array/SubscriptArray.h

```
template<Dimension ndim>
class SubscriptArray {
    // Empty template -- supply a specialization for each
    // value of ndim.
};
```

This template has an empty body, and its only purpose is to specify that names of the form SubscriptArray<ndim> are template class names. This done, we can define specific classes with such names:

Array/SubscriptArray.h

```
class SubscriptArray<1> {
public:
    SubscriptArray() {}
    SubscriptArray(Subscript s0) : sub0(s0) {}
    SubscriptArray<1>& operator=(Subscript rhs) {
        sub0 = rhs;
        return *this;
    }
    Subscript& operator()(Subscript)       { return sub0; }
    Subscript  operator()(Subscript) const { return sub0; }
private:
    Subscript sub0;
};

class SubscriptArray<2> {
public:
    SubscriptArray() {}
    SubscriptArray(Subscript s0, Subscript s1) {
        sub[0] = s0;
        sub[1] = s1;
    }
    SubscriptArray<2>& operator=(Subscript rhs) {
        sub[0] = sub[1] = rhs;
        return *this;
    }
    Subscript& operator()(Subscript i)       { return sub[i]; }
    Subscript  operator()(Subscript i) const { return sub[i]; }
private:
    Subscript sub[2];
};
```

Except for their names, these two classes are independent: Their constructors and member functions have different numbers of arguments, and there is nothing common about their implementations. Likewise there is no connection between the types SubscriptArray<1> and SubscriptArray<2>.

Nevertheless, the common form of their names can be exploited. Suppose we want to write a class template parameterized by dimensionality that has a SubscriptArray<ndim> as a member, like this:

ch12/tSubscriptArray.C

```
template<Dimension ndim>
class UsesSA {
public:
    // ...
private:
    SubscriptArray<ndim> a;
};
```

All classes generated from UsesSA<ndim> have a SubscriptArray<ndim> member, and those members are different for different values of ndim. A practical example of this usage appears on page 389.

An explicit class definition with a template class name for a name (like SubscriptArray<1>) is called a *template specialization* because it is a special case of a more general template. In the example we've used here, the class template itself doesn't have any content. In other cases, the class template might supply an implementation suitable for most values of the template parameters, and a template specialization could be supplied for others.

The important point is this: A class template does not define a type. A template class generated from a class template defines a type, as does a template class defined explicitly (i.e., by a template specialization).

Having established that a class template is not a type, we need to look at the role of function templates as clients of template classes. Consider the tuplize() function template on page 305. We need not think about SimpleArray<int>, SimpleArray<float>, or any other specific instance of the template. As a client of the template, we tend to think of the common parameterized specification as if it were a type. This view is convenient and not harmful as long as you remember that type checking is always done on the template classes. C++ will not prevent us, for example, from writing a template specialization for SimpleArray<float> that does not have an operator[]() member. Were we to do that, calling tuplize() with a SimpleArray<float> would fail (at compile time).

Since C++ type checks templates only after expansion, a class template is a weaker specification than an interface. In the worst case, a class template guarantees nothing other than the form of the template class name, whereas an interface guarantees that the class will have certain member functions each with a specified name, argument type, and return type. More commonly, template classes are

generated from their corresponding class templates, and one can count on name commonality among all the generated template classes. Generally it is a good idea to avoid surprises:

- Where possible, template specializations should provide name commonality with their corresponding class templates.

12.5.2 Types from Templates with Inheritance

On page 309, we derived publicly CheckedSimpleArray<T> from SimpleArray<T>. Therefore an instance of the type CheckedSimpleArray<int> contains an instance of the type SimpleArray<int> as a base subobject. The public derivation allows initializing SimpleArray<int> references with references to CheckedSimpleArray<int> objects. No type relation is built between SimpleArray<int> and, say, CheckedSimpleArray<float> or any other class created from the template.

Despite there being no type relation between different template classes generated from a common class template, there is a relation in the template itself. The class template CheckedSimpleArray<T> is derived from the class template SimpleArray<T>. Thus function templates, like tuplize() on page 305, that act on references to the templatized base class, const SimpleArray<T>& , can be applied to instances of the templatized derived class. Conceptually there is a single tuplize() function that operates on any reference to any SimpleArray<T> object. Public derivation of CheckedSimpleArray<T> from SimpleArray<T> allows tuplize() to apply to CheckedSimpleArray<T> objects.

Once again, C++ provides no guarantee that this conceptually simple model always holds. One could define a template specialization for, say, CheckedSimpleArray<float> that was not derived from SimpleArray<float>. In that case, tuplize() could not be applied to an instance of SimpleArray<float>. Avoid surprises:

- Template specializations should provide the same type relationships as the corresponding class templates.

12.5.3 Types from Templates with Interfaces

The preceding discussion on templates and public inheritance applies equally well to public derivation from templatized interfaces. For example, the sum() function on page 320 applies to any object from any class derived from the templatized interface Array2d<T>. The template and interface mechanisms work together when a creation type is passed to sum(), as in this code:

ch12/tSum.C

```
FormedArray2d<double> f(5, 5);
f = 4;
double d = sum(f);
```

The function sum() was written for the template Array2d<T>. When f is defined, the FormedArray2d<double> instance of the template causes the compiler to instantiate an Array2d<double> interface. Calling sum() with f sets T inside sum() to double and binds the resulting Array2d<double>& function argument to f's interface. For Array2d<double>, the return type of the subscript operator is double, allowing the += operator to match types and sum the double objects.

The similar code

ch12/tSum.C

```
RigidArray2d<double,5,5> f(5, 5);
f = 4;
double d = sum(f);
```

creates a completely different type for this f, RigidArray2d<double, 5, 5>. Both this rigid array and the formed array in the preceding example share a common interface base class Array2d<double>, and both bits of code call the same sum() function, the function for Array2d<double>.

On the other hand, the similar code

ch12/tSum.C

```
FormedArray2d<int> f(5, 5);
f = 4;
int d = sum(f);
```

shares only the source code for sum() and the source code for the declarations in Array2d<T>. Array2d<int> is the interface in this case, and a different sum() function must be generated by the compiler to match this interface.

We can choose to think about the template instances created for these examples, allowing us to estimate the number of different sum() functions that will be created by the compiler, for example. We can also choose not to think about the instances. In all three of the preceding examples, it looks like exactly the same sum() function was applied. This freedom from the details of correctly matching the types of the summer and the summands, combined with the security of knowing that these types do in fact match, is the power of abstraction.

12.5.4 Types of Template Parameters

A template parameter declared as class T specifies that T is a type. Unlike the declaration class T appearing outside of a template parameter list, T need not be a class. Built-in types, structs, and typedef types can also appear as type values in template class names. Thus the class T parameter in CheckedSimpleArray<T> on page 309 can be bound to anything that is a type:

ch12/tCheckedSimpleArray.C

```
CheckedSimpleArray<int> ia(4);                    // ok
CheckedSimpleArray<int*> ipa(4);                  // ok
```

```
typedef float Coord;
CheckedSimpleArray<Coord> ca(4);                    // ok
CheckedSimpleArray<float> fa(4);                    // ok, same as previous.

CheckedSimpleArray< CheckedSimpleArray<int> > iaa(4);   // ok
```

Note that the typedef is "unraveled" before the template is instantiated, so ca and fa have identical types.

As we discussed in Section 11.4.4, template parameters need not be types. For example, constant expressions that evaluate to integer values can also be used. The only type-related issue here is that expressions are evaluated before a class template is instantiated. Thus

ch12/tObjtype.C

```
const int nrows = 3;
RigidArray2d<int, nrows, 2*nrows> b;
RigidArray2d<int, 3, 6> b_too;
```

creates two arrays of the same type.

12.6 Restricted Template Expansion

We have reanalyzed most of the examples from the preceding chapters, focusing on the issues of type and type relationships. We introduced one new type relationship, the are-used-together relation in Section 12.4. Now we introduce a second new relationship and show how it can be expressed in C++.

When a subset of functions in a group of classes are related to each other in a structurally similar way, we say they share *function structure*. For example, in most classes, != should be related to == by logical negation. One way to achieve this is a template function for !=:

ch12/not-equal.C

```
template<class T>
Boolean operator!=(const T& lhs, const T& rhs){
    return ! (lhs == rhs);
}
```

This expresses the function similarity, but it does so for all possible classes T. This function template forces there to be a != defined for every class with ==. Moreover, the only way to control the definition of != for new classes is to overload its definition for each new class. Any new function template for != will either conflict with this one or be weaker and thus not be applied by C++ when function matching is performed (see Section 11.5). Using a function template to express the

function-structure commonality is too general because the type T is not restricted to be related to the class used in the definition of ==.

This reasoning leads us to consider an implementation base class: Then the argument type for the == operation would be related to the argument type for the != and the scope of the function definition would be limited to the base class. However, the argument types will need to be different for different classes. For example, two objects in the GPIBInstrument category may be compared for equality by comparing GPIB addresses, whereas two Array<T> objects are compared for equality using element-by-element comparison.

Suppose we try to form a common base class for these two classes that provides the common function structure (namely, that implements the inequality test as the logical negation of the equality test). Let's call the common class Object and include a virtual function for == to be implemented by each derived class. The argument for this function must be an Object reference. The right-hand side of the == may be a GPIBInstrument or an Array<T>, but Object only knows about their common features. This approach fails because the derived-class object—GPIBInstrument or Array<T>—cannot compare itself to an Object. Any reasonable equality test is type dependent.

To obtain both type-restricted and type-dependent functions, we combine the features of implementation and template categories. Specifically, we create an implementation base class, EquivalentCategory<DerivedType>, parameterized by the type of the derived class. The base class provides the common implementation forcing != to be logically opposite to ==; the template parameter gives these functions the correct type and gives access to a type-dependent definition for ==.

As we are using both class and function templates with inheritance and type conversions, let's dig into an example. Here is the parameterized base class that implements two operators, == and !=:

SciEng/EquivalentCategory.h

```
template<class DerivedType>
class EquivalentCategory {
    // This class needed for type restriction
    friend Boolean operator==(const DerivedType& lhs, const DerivedType& rhs) {
        return lhs.equivalentTo(rhs);
    }
    friend Boolean operator!=(const DerivedType& lhs, const DerivedType& rhs) {
        return ! lhs.equivalentTo(rhs);
    }
};
```

The operators are global template functions, but matching actual arguments are limited to types that can be used to initialize const DerivedType references. The sole

purpose of this class template is type restriction. When the class template is expanded for a particular DerivedType, the global function is declared only for arguments of type DerivedType.

Here is one of those derived classes:

ch12/tEquality.C

```
template<class T>
class ComparableFormedArray1d :
    public FormedArray1d<T>,
    public EquivalentCategory< ComparableFormedArray1d<T> > {
public:
    ComparableFormedArray1d(Subscript n);
    ComparableFormedArray1d(const FormedArray1d<T>& f);
    // ...

    // User Must Define for EquivalentCategory
    Boolean equivalentTo(const ComparableFormedArray1d<T>&) const;
};
```

Notice that we have used the name of the derived class itself as the parameter for the base class. With this class we can write:

ch12/tEquality.C

```
ComparableFormedArray1d<int> a(4);
ComparableFormedArray1d<int> b(4);
// Set elements of a and b...

if ( a == b ) {
    // ...
}
// ...
if ( a != b ) {
    // ...
}
```

A similarly defined ComparableRigidArray1d<T,n> could be used in similar code, and yet our global template functions will not expand for arbitrary types. Thus we have both restricted the application of these global operators and provided definitions that are consistent across all classes that use the EquivalentCategory<DerivedType> template.

The definitions of the global operators in the base class are given in terms of a function of the derived class, equivalentTo(). The derived class must supply this function. For example, ComparableFormedArray1d<T> might define it thus:

ch12/tEquality.C

```
template<class T>
Boolean ComparableFormedArrayId<T>::
equivalentTo(const ComparableFormedArrayId<T>& rhs) const {
    if (shape(0) != rhs.shape(0) ) return Boolean::false;
    for (Subscript i = shape(0)-1; i >= 0; i--) {
        if ((*this)(i) != rhs(i)) return Boolean::false;
    }
    return Boolean::true;
}
```

We call this kind of function a "user-must-define" function for EquivalentCategory<DerivedType>: The user, ComparableFormedArrayId<T> for example, must provide this function when deriving from the template base class.

Another interesting aspect of this example is the implicit definition of the global template functions for the operators == and !=. The explicit declaration of ComparableFormedArrayId<int> creates instances of the array template and of the EquivalentCategory<DerivedType> template with DerivedType set to the array type. The generated base class contains definitions of the global operators as friend functions. C++ applies the rules of ordinary global function declarations when == and != appear in expressions with ComparableFormedArrayId<T> objects. They are not treated as template functions for matching (see Section 5.6), and yet the template function definitions will be used to define these functions if they are called.

12.7 Function-Structure Categories

The mechanism we introduced in the previous section can be viewed as more than a trick to control global template expansion. If we analyze the overall result of the system built around EquivalentCategory<DerivedType>, we conclude that it established the rule that (a != b) is equivalent to !(a == b) for any two instances a and b of a class. This is a kind of commonality unlike others we have examined before, a commonality of function structure.

We call the system that expresses function structure commonality a *function-structure category*, alluding to the sharing of systematic function definition rules. The code for EquivalentCategory<DerivedType> is written once and shared among all classes that derive from this template. These derived classes form a category of classes sharing the code in the template base class.

Classes in function-structure categories share common structure in their member functions. For example, classes in EquivalentCategory<DerivedType> share the relation that == and != always give logically opposite results. The common behavior is expressed by functions in the category base class, like functions for the == and != operations. These functions then use functions provided by classes in the

category the "user must define" functions, such as the equivalentTo() function on page 354.

Unlike the classes in an interface category, which are all derived from a single interface base class, the classes in a function-structure category are each derived from a different base class (namely, the class stamped out from the parameterized base class template). Consequently a function-structure category does not provide type relationships among the category members, only between each category member and its own base class.

A function-structure is a parameterized base class. Like interface categories, function-structure categories are not concerned with and do not use member data directly; unlike interface categories, the argument and return types of the member functions depend on the derived class, not just the interface. Like template categories, function-structure categories are parameterized to express common structure; unlike template categories, function-structure categories only specify some relationships among functions that can be shared across many classes; function-structure categories do not specify complete classes.

The example of EquivalentCategory<DerivedType> is not compelling because it expresses only a small amount of function structure. Chapter 16 provides an extensive example using function-structure categories to build a system of classes for abstract algebra.

12.8 Summary

Types link together all aspects of C++. The daily activities of a C++ programmer are centered around writing, relating, and using types. We use built-in types to create programmer-defined types: classes. We use derivation and templates to create programmer-defined type relationships: interface, implementation, and template classes. We express our algorithms and design systems of interacting objects using types.

Type checking requires that all of the uses of types agree with the definitions of those types. This means that function calls that use objects through interfaces are checked and that objects are checked before they can be used to initialize interface references. The intimate connection between the type-checking rules of C++ and the rules of templates means that as a function call is checked, matches to templates cause them to be applied automatically.

Using types correctly is enforced by C++. Using types to express commonality is our responsibility. C++ only helps in continuing to insist that new types introduced to express commonality are related consistently. As your experience with C++ grows, you will come to rely on this help more and more. Eventually designing classes to cause C++ to trap as many programming errors as possible—designing robust code—becomes one of the central goals of programming.

	Types	
	Built-in	*Programmer-defined*
Creation	int, float	Point, Acme130_VS
Use	const int& , double*	Point& , VoltageSupply&
Interface	(none)	VoltageSupply, Array1d<int>
	Type Relations	
Parametric	int ⇒ const int, int ⇒ int*	SimpleArray<T>, Fallible<T>
Conversion	int ⇒ double	Acme130_VS& ⇒ VoltageSupply&, Fallible<int> ⇒ int
Function-structure	a!=b means !(a==b)	EquivalentCategory<DerivedType>
Has base of	(none)	CheckedSimpleArray<T> ⇒ SimpleArray<T>
Are used together	(none)	(Acme130_GC+GPIBController) ⇒ GPIBComponent

Table 12.1 Examples of Types and Type Relationships Discussed in this Chapter.

Learning the large number of potential relationships that C++ supports and harnessing them to create robust programs is a challenge that must be met through experience. Table 12.1 summarizes the relationships we have discussed in this chapter.

In the same way that sophisticated systems must be built by iteration and revision, learning the complete C++ type system can only be done by writing and rewriting programs in C++. Thus we end our description of the C++ language with this chapter on types only to start a new part of the book focused on extended examples. We hope that by examining these examples you can experience some of the ways that C++ can be used in scientific and engineering programming.

12.9 Notes and Comments

12.1 What we call built-in parameterized types are examples of what Ellis and Stroustrup call "derived types" [44, Section 3.6.2]. We prefer the term parameterized to avoid confusion with class derivation and to connect these types with templates.

12.2 Not all type errors can be caught at compile time: A C++ object can be in a partially constructed state during which time some of the operations allowed by its type may not be valid. Such errors may not be caught by the compiler. In particular, pure virtual functions may not be called, either directly or indirectly, from a constructor of an abstract base class. Consider the following code:

ch12/pure-virtual-called.C

```
class A {
public:
    A() { init(); }
    virtual void f() = 0;
    void init() { f(); }
};

class B :
    public A {
public:
    B() : A() {}
    virtual void f() { }
};

int main() {
    B b;
    return 0;
}
```

The constructor for B invokes the constructor for A, which calls init(), which then calls the pure virtual function f(). Since the B object is only partially constructed, f() is not yet defined, even though f() is one of the operations allowed by the types A and B. See also [44, Section 10.3].

12.3 A type can also be defined as the set of all objects that have a certain behavior. Such a view—not taken by C++—leads to a language without programmer specification of typelike classes. After all, we should be able to let the computer examine a function to see what behaviors it requires for the objects it manipulates and compare that to the behaviors expressed by objects passed in as arguments. As attractive as this proposition may sound, the information content of type specifications allows more efficient programs than has yet been possible in languages designed without them.

12.4 Our description of interface or use types in terms of a partial specification of behavior matches the substitutability of types principle in [115] and [86]. The create/use dichotomy evident in C++ and other languages is being formalized in new theories of type, for example [75]. (These works attempt to systematize types in a wider context than static, sequential object-oriented programming.) The foundations of the Axiom computer algebra system [61] provide an interesting view of parameterized types.

12.5 Encapsulation can be viewed as creating another kind of type relationship. Consider the following code:

ch12/tTypeErrsEg.C

```
SimpleArray<int> a(5);
SimpleArray<int> b(5);
a.copy(b);    // Compile Error!
```

The member function copy() (page 101) is part of the specification of SimpleArray<T>, but it is private. Calls to the function from outside of the class are illegal. Non-const functions are illegal when called against const objects and private functions are illegal when called from outside of the class scope. In a limited way, the public member functions form a built-in parametric subtype for a class.

Despite this parallel, viewing private as a built-in parametric type has limited value. The C++ programming activities that require a thorough understanding of type, especially programming through interfaces and with templates, rely on declaring objects or references. We cannot declare an object or a reference to have something like a private type. Thus the rules for encapsulation are most simply stated by saying that private objects, references, pointers, and functions cannot appear in public expressions.

12.6 Better terms for base class and base type would be subclass and subtype, respectively, meaning a class (type) whose operations are a subset of the operations of another type. Unfortunately, as noted by Ellis and Stroustrup [44, Section 10], *subclass* has been used previously to mean what C++ calls a derived-class type. Using base type here helps connect to standard C++ terminology, but it does not convey the critical idea of type subset as well as subclass and subtype would.

12.7 The interplay between encapsulation, inheritance, and types was elucidated in a readable paper by Snyder [100].

12.8 Stroustrup discusses the disadvantages of fat interfaces in [107, Section 13.6], concluding that they "are best avoided where run-time performance is at a premium, where strong guarantees about the correctness of code are required, and in general wherever there is a good alternative" (page 454).

12.9 We used to call our function-structure category a *mix-in* category. Booch [16, page 113] defines a mix-in to be "a class that embodies a single, focused behavior, used to augment the behavior of some other class via inheritance." The term "mix-in" is common in LISP-based object-oriented languages such as CLOS and Flavors. However, the term is vague and we wanted a name for the derived-class parameterized base class construct that implements function-structure commonality in C++. Moreover, the C++ technique we call base-class composition has been called a "mix-in" at least informally. Thus we chose a new name.

12.10 Exercises

12.1 Draw the class DAG for the base classes of FormedArray1d<float>, RigidArray1d<float>, FormedArray1d<int>, and RigidArray2d<int>. Identify all of the interface base classes common to more than one of these classes.

12.2 Virtual member functions increase the potential for code reuse by allowing derived classes to override function definitions provided by base classes. It would seem, therefore, that maximum reuse would be obtained if C++ made all member functions virtual. Suggest reasons why C++ does not make member functions virtual by default.

12.3 Give a practical example in which the rule that temporary objects are considered const enables C++ to detect an "obvious" programming error.

12.4 Starting with the GPIBController_GC class on page 247, write a GPIBController_GC_GC version of the class that can be put on a list of GPIBComponent*.

12.5 Write a DispatchingList<T, AcceptorType> class template that is a list of T* elements with a dispatchToAll() member function taking a reference to an instance of a class in the AcceptorType interface category. This function should dispatch to each element on the list, passing the acceptor. Use this class template to reimplement the printing loop on page 345.

12.6 Suppose that you purchased a GPIB components class library that looks like ours, but that you are prevented by licensing restrictions from modifying any of the code. Whoever wrote the classes did not anticipate your need to work with lists of GPIB components as we have in Section 12.4, so no GPIBComponent class is provided. Design and implement a set of classes that lets you work with lists of GPIB components. Do not use type casts.

12.7 Consider the following code:

ch12/tMixApplesOranges.C

```
class Apple :
    public EquivalentCategory<Apple> {
public:
    Apple(int n) : a(n) {}
    virtual Boolean equivalentTo(const Apple& an_apple) const { return a == an_apple.a; }
private:
    int a;
};

class Orange :
    public EquivalentCategory<Orange> {
public:
    Orange(int n) : o(n) {}
    virtual Boolean equivalentTo(const Orange& an_orange) const { return o == an_orange.o; }
private:
    int o;
};

int main(){
    Apple an_apple(1);
    Orange an_orange(1);
    if (an_apple == an_apple && an_orange == an_orange) return 0;
    return 1;
}
```

Try to replace the EquivalentCategory<DerivedType> function-structure category with an interface base class having a definition for equivalentTo() such that this code compiles with only the changed base classes. Can you see that you cannot succeed?

12.8 C++ does not allow a catch with a base class argument to precede a catch with a derived-class argument. Suggest a reason for this rule.

12.9 Design, implement, and test a function-structure category that defines all the comparison operators (==, !=, <, <=, >, and >=) in terms of a single user-must-define function.

PART III

Applications and Techniques

CHAPTER 13

Arrays

This chapter shows how the concepts and C++ mechanisms already discussed can be used to define a system of array classes. We present the development of these classes in considerable detail, both to illustrate the practical application of the techniques discussed in Part III and to develop array classes that we can use as a basis for other examples in subsequent chapters. To allow nonnumeric array elements, our arrays are not endowed with numerical operations like dot product and matrix multiply; we treat such operations in Chapter 16.

Since arrays are critical to many scientific and engineering programs, we have chosen adaptability and convenience of use over simplicity of construction. Adaptability—the ability to trade off performance, flexibility, and ease of reuse—dictates a consistent approach across a wide range of array types. Convenience of use—the availability of expected operations and compatibilities—dictates that the range of array types include those useful with FORTRAN, C, and object-oriented C++. We sacrifice simplicity of construction to achieve these aims: This is a long chapter with many uses of advanced C++ features.

The array system is divided into two major subsystems, each with different design goals. The classes in our system of *concrete array* classes gain both space and time efficiency by omitting virtual function interfaces. At the same time, omitting virtual function interfaces makes some kinds of programs harder to write and maintain. Our *interfaced array* classes provide virtual function interfaces in the style we introduced in Chapter 9, sacrificing efficiency to gain other benefits. This partition of our array system represents one of the trade-offs we seek to express.

Our concrete array classes are designed for maximum speed and minimal space, while at the same time achieving reuse through templates and name commonality. If implemented carefully, a concrete class can provide compute time performance comparable to implementations in FORTRAN or C of the same function. Concrete classes are also important when even a small space overhead is significant. For example, in a computer-aided design system, an array of three floating point numbers might be used to store the coordinates of a point. Since there might be tens or even hundreds of thousands of points in such a program, an overhead

of one or two words of storage per point, as would be imposed by using virtual functions, might be unacceptable. Even while we aim for these runtime performance goals, we must also provide compile time safety and convenience of use for programmers.

Our interfaced array classes rely on interface base classes to allow a client function to work with different array implementations; they minimize source code development and compiled-code size at the expense of runtime object space and member function call overheads. The interfaced array classes we develop in this chapter expand on the classes we sketched in Section 11.3. Most significantly, we add *array projection* classes, which provide arraylike access to subsets of the elements of an array. For example, array projection classes can be used to provide access to specified rows or columns of a two-dimensional array.

We will also introduce *iterators*, objects that step through the elements of an array (or, more generally, of any collection of objects; cf. Section 6.10). Iterators generalize the usual looping over arrays, avoiding the random access implied by subscripting. They are useful in some whole-array operations independent of subscripts, like adding a constant to all elements of a numerical array.

Throughout this chapter, we will use the terminology defined in Section 11.3. It might be helpful to reread that section before continuing with this chapter.

13.1 Using Concrete Arrays

We begin with the concrete array classes. Before describing how to implement the concrete array classes, let's look at some examples of how they can be used.

Table 13.1 lists the classes intended for direct use in client code (as opposed to classes that provide a framework for developing new array classes). We use the prefix Concrete to indicate those classes designed for minimal overhead. We use the words Rigid and Formed to indicate that the array size is fixed at compile time (Rigid) or set when the array is created but changeable (Formed). For the reasons described in Section 11.4.2, we also incorporate an array's dimensionality in its class name. This systematic naming process yields long names; if you like shorter names, use typedef statements to define abbreviations.

Our aim is a family of array classes designed to exploit name commonality so that we can write client code that will work with a variety of arrays. For example, the following function template can find the maximum element of an array object if that object provides a typedef EltT for the element type (see Section 11.4.3), a shape() member for obtaining the array shape, and a two-argument subscripting function using the parentheses or function call operator:

Class Name	*Storage Layout*
ConcreteFormedArray1d<T>	Row major, contiguous
ConcreteRigidArray1d<T, n0>	Row major, contiguous
ConcreteFortranArray1d<T>	Column major, contiguous
ConcreteFormedArray2d<T>	Row major, contiguous
ConcreteRigidArray2d<T, n0, n1>	Row major, contiguous
ConcreteFortranArray2d<T>	Column major, contiguous
ConcreteFortranSymmetricPackedArray2d<T>	Upper triangular, packed
ConcreteFormedArray3d<T>	Row major, contiguous
ConcreteRigidArray3d<T, n0, n1, n2>	Row major, contiguous
ConcreteFortranArray3d<T>	Column major, contiguous

Table 13.1 Concrete Array Classes Intended for Use by Client Functions

ch13/tConcreteMax.C

```
template<class ArrayType>
ArrayType::EltT maxArray2dElement(const ArrayType& a) {
    ArrayType::EltT the_max = a(0, 0);
    for (Subscript i = 0; i < a.shape(0); i++) {
        for (Subscript j = 0; j < a.shape(1); j++) {
            the_max = max( the_max, a(i, j) );
        }
    }
    return the_max;
}
```

The expression a(i, j) illustrates array subscripting with parentheses, FORTRAN style, but 0-origin indexing, C and C++ style. (See Exercises 13.8 and 13.9.) The function template max() (not shown) returns the maximum of its two arguments.

The preceding function template can work on a variety of arrays provided that they supply the required functions with the required names. Our array classes provide a consistent set of names. For example,

ch13/tConcreteMax.C

```
ConcreteFormedArray2d<int> a1(3, 4);          // 3x4 array of ints
ConcreteRigidArray2d<float, 5, 6> a2;         // 5x6 array of floats
ConcreteFortranArray2d<double> a3(7, 8);      // 7x8 array of doubles
// ... set elements ...
cout << "max element of a1 is " << maxArray2dElement(a1) << endl;
cout << "max element of a2 is " << maxArray2dElement(a2) << endl;
cout << "max element of a3 is " << maxArray2dElement(a3) << endl;
```

uses identical calls to maxArray2dElement() even though we have used different storage for the elements in all three cases.

The maxArray2dElement() template accesses array elements with the function call or parentheses operator. The other important ways of accessing elements are by whole-array assignment, projection onto lower dimension, and sequential access of array elements independent of array shape. We give some examples of the first two access methods here; sequential access is described in the section on iterators, Section 13.7.

Assignment to an array, either from a single value or from an array of the same shape, sets the values of all the array's elements:

ch13/tConcreteMax.C

```
ConcreteFormedArray1d<float> f1(5);
ConcreteFormedArray1d<float> f2(5);
f1 = 130.0;
f2 = f1;
```

Here all elements of both arrays f1 and f2 are set to 130.0. Size mismatches on array-to-array assignment are caught at compile time for rigid arrays and at runtime for formed arrays. See also Exercise 13.3.

A lower-dimensional cross-section of an array, called an array *projection*, can be obtained using the project() member function. The result of project() is the cross-section of the array obtained by holding the subscript specified by the second argument constant at the value specified by the first argument and letting the other subscript(s) vary across their range. For example

ch13/tConcreteMax.C

```
ConcreteFormedArray2d<int> a(3, 4);
a(0,0)=1;    a(0,1)=2;    a(0,2)=3;    a(0,3)=4;
a(1,0)=5;    a(1,1)=6;    a(1,2)=7;    a(1,3)=8;
a(2,0)=9;    a(2,1)=10;   a(2,2)=11;   a(2,3)=12;

cout << a.project(0, 0) << endl;    // Prints: [1, 2, 3, 4]
cout << a.project(1, 0) << endl;    // Prints: [5, 6, 7, 8]
cout << a.project(2, 0) << endl;    // Prints: [9, 10, 11, 12]

cout << a.project(0, 1) << endl;    // Prints: [1, 5, 9]
cout << a.project(1, 1) << endl;    // Prints: [2, 6, 10]
cout << a.project(2, 1) << endl;    // Prints: [3, 7, 11]
cout << a.project(3, 1) << endl;    // Prints: [4, 8, 12]
```

For a two-dimensional array, a second argument of 0 yields a row of the array, whereas a second argument of 1 yields a column. More generally, if the second argument of project() is i, we say that we are projecting along dimension i. When

we define two-dimensional concrete arrays in Section 13.2.4, we provide row() and column() members that call project() with second arguments of 0 and 1, respectively.

For both conciseness and compatibility with C++ built-in arrays, we also define the [ ] operator to be equivalent to projection along dimension 0. Thus

ch13/tConcreteMax.C

```
cout << a[0] << endl;          // Prints: [1, 2, 3, 4]
cout << a[1] << endl;          // Prints: [5, 6, 7, 8]
cout << a[2] << endl;          // Prints: [9, 10, 11, 12]
```

Since C++ calls square brackets the subscript operator, it might seem that using square brackets for projection is potentially confusing. However, C++ offers subscripting only by repeated projection: The (i, j)th element of a built-in array a is accessed by a[i][j], where a is a one-dimensional array of one-dimensional arrays, a[i] selects the ith subarray, and [j] selects the jth element of the subarray. The square brackets operator for our arrays cascades in the same manner:

ch13/tConcreteMax.C

```
cout << a[1][2] << endl;   // Prints: 7
```

The first square brackets operator selects the one-dimensional subarray row 1, and the second square brackets operator selects element 2 because projection on a one-dimensional array yields a zero-dimensional array (i.e., an array element).

A projection refers to the elements in the array; no copy is made. Thus

ch13/tConcreteMax.C

```
a.project(0, 1) = 1;
```

sets the first column of the array a to 1 and

ch13/tConcreteMax.C

```
a.project(0, 0) = 2;
```

or

ch13/tConcreteMax.C

```
a[0] = 2;
```

sets the first row of a to 2. In each case, a temporary object is created from the expression on the left-hand side of the equals sign. In the terminology of C++ reference manuals, the object returned by project() is an *lvalue*, meaning that it is legal on the left-hand side of an assignment. The lvalue projection object acts like a lower-dimensional reference into an array.

Exactly what type of temporary object is returned by project()? To achieve adaptability and convenience of use, we deliberately avoid programming with the explicit type of the projection of an array. Instead we describe its properties—a lower-dimensional, arraylike reference object—and we give its type as a name in each array class. Each array class provides two type names (each either a nested class or a typedef), ProjectionT and ConstProjectionT, that specify the types of the objects returned by the project() member functions. When called on a const array

object, project() returns a ConstProjectionT object; when called on a non-const array object, project() returns a ProjectionT object.

Here is an example that needs the type name to return a lower-dimensional object:

ch13/tConcreteMax.

```
template<class ArrayType>
ArrayType::ProjectionT maxArray3dPlane(ArrayType& a) {
    // Returns plane of a that contains the maximum value
    Subscript plane_with_max = 0;
    ArrayType::EltT max_value = a(0, 0, 0);
    for (Subscript plane = 0; plane < a.shape(0); plane++) {
        ArrayType::EltT max_value_in_plane = maxArray2dElement(a[plane]);
        if (max_value_in_plane > max_value) {
            plane_with_max = plane;
            max_value = max_value_in_plane;
        }
    }
    return a[plane_with_max];
}
```

This function takes a three-dimensional array and returns the plane of the volume that contains the maximum element. The return type is specified to be an object of whatever type results from calling project() on a non-const array.

To use the returned object, we don't need to know its type, only that it has arraylike behavior and that it refers to elements of the array for which project() was called. Thus the code

ch13/tConcreteMax.C

```
ConcreteFormedArray3d<int> fa(2, 3, 4);
// ... set fa ...
maxArray3dPlane(fa) = -1;
```

sets the plane of fa that contains the maximum value in fa to −1.

13.2 Concrete Arrays

In the previous section, we sketched a family of array classes by giving examples of their use. In this section, we turn to their implementation. Table 13.2 summarizes the behavior each concrete array class should provide.

The design of these array classes must address three major challenges:

Variety of storage layouts. We want a consistent system of arrays compatible with common storage layouts used in FORTRAN, C, and C++ programs.

Name	*Description*
dim()	Dimensionality
shape(Dimension d)	Shape in dimension d
numElts()	Total number of elements
EltT	Type of elements
BrowserType	Type of browser over all elements
IteratorType	Type of iterator over all elements
ProjectionT	Type of read/write projection
ConstProjectionT	Type of read-only projection
operator()(Subscript...)	Subscript access to element
project(Subscript, Dimension)	Projection along dimension
operator[ ](Subscript)	Projection along dimension 0
row(Subscript)	Projection along a row (2D and 3D)
column(Subscript)	Projection along a column (2D and 3D)
plane(Subscript)	Projection along a plane (3D)

Table 13.2 Functions and Names Provided by Concrete Array Classes. Functions listed in the top portion are element-type independent.

Projections. Each array class must provide projection objects that act like references to selected elements of array objects.

const *arrays.* Read-only access to elements of const arrays and to elements of projections of const arrays is needed.

The storage layout variety challenge is met in Section 13.2.1 using a template parameter, called Subscriptor, that specifies the storage layout. The const and projection challenges are coupled: The const operations on a projection must be consistent with operations on a const projection. To meet these challenges, Section 13.3 describes objects that are array references, in both const (ConcreteArray2dConstRef) and non-const (ConcreteArray2dRef) versions. Then Section 13.4 uses these reference objects to implement array projections. By facing all of these challenges, our arrays will work in a wide variety of cases.

13.2.1 Storage Layout

We want a consistent family of concrete array classes with various element storage layouts. In particular, we want to support arrays that store their elements compatibly with FORTRAN, C, and C++ built-in arrays. C and C++ built-in arrays are identical but different from FORTRAN built-in arrays. We also want to be able

to implement various packed array formats, such as those used by LAPACK [5] to store symmetric and triangular matrices. We treat the FORTRAN and C cases here and a LAPACK-compatible, packed layout in Section 18.5.

FORTRAN and C++ built-in arrays are *strided arrays*, arrays in which the number of memory locations between successive elements of the array is constant. In FORTRAN, the storage is laid out in *column-major* order: The leftmost subscripts vary fastest. For a 3×4 array, the layout is

$A_{0,0}$	$A_{1,0}$	$A_{2,0}$	$A_{0,1}$	$A_{1,1}$	$A_{2,1}$	$A_{0,2}$	$A_{1,2}$	$A_{2,2}$	$A_{0,3}$	$A_{1,3}$	$A_{2,3}$

In C++, the storage is laid out in *row-major* order; the rightmost subscripts vary fastest, leading to this layout:

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$

The *stride* in any dimension is the number of elements between the beginning of one element and the beginning of the next element in a given dimension. For the 3×4 matrix in the column-major layout, the stride for dimension 0 is 1 and the stride for dimension 1 is 3; for the row-major layout, the strides are 4 and 1.

(For those familiar with the use of FORTRAN arrays in subroutine libraries, strides are related to the common practice of passing to a subroutine the leading dimension of an array. The leading dimension gives the stride between column origins; when the leading dimension is larger than the size of the array the subroutine is to work with, the stride gives the subroutine the information it needs to compute element locations.)

Like FORTRAN and C++ built-in arrays, the storage for our arrays is a contiguous block of objects of one type. The mapping from subscripts to objects is more general, however, allowing row or column-major storage and many kinds of packed arrays. The requirement is that any array element can be accessed given a pointer to contiguous storage and an integral offset computed as a function of the indices.

The concrete array class templates (and function templates that accept instances of these classes) are parameterized by a Subscriptor class. A Subscriptor provides a member function named offset(), which takes subscripts as its arguments and returns the offset to the corresponding element. Particular array classes substitute the Subscriptor parameter with classes like ConcreteColumnMajorSubscriptor<2>, for column-major subscripting of two-dimensional arrays, or ConcreteRowMajorSubscriptor<3>, for row-major subscripting of three-dimensional arrays. Each supplies an appropriate offset function as well as the element-type independent functions listed in Table 13.2.

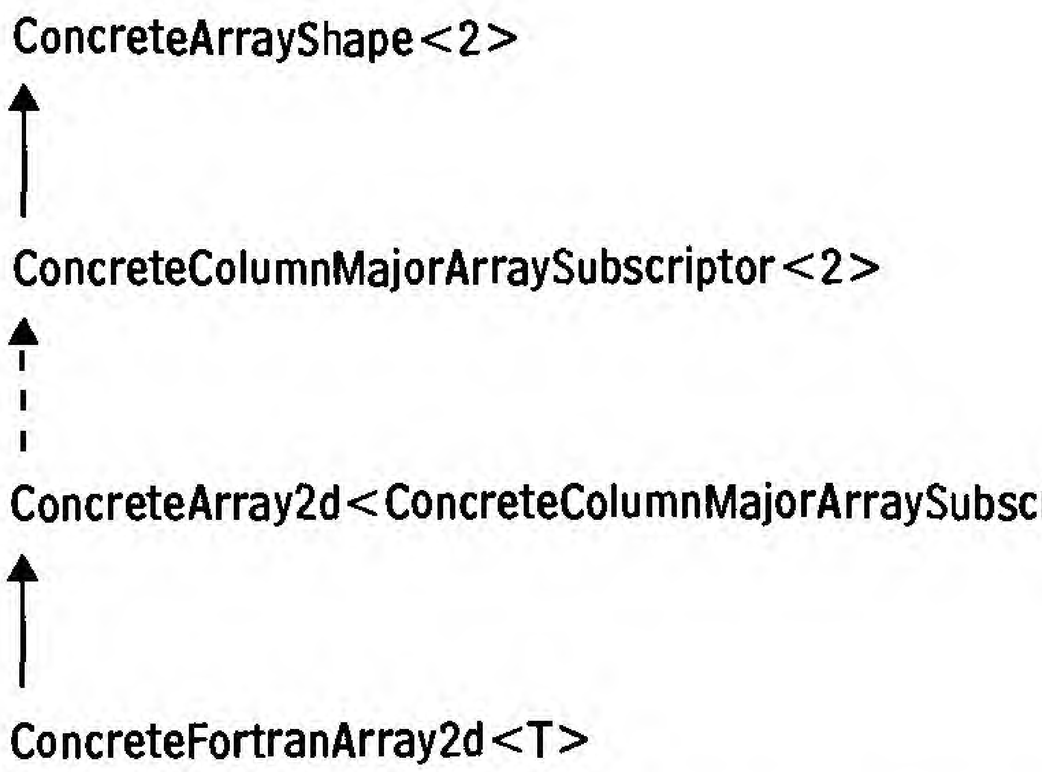


Figure 13.1 Class DAG for ConcreteFortranArray2d<T>. The dashed arrow indicates private derivation.

In the following subsections we use ConcreteFortranArray2d as our running example, implying that we use ConcreteColumnMajorArraySubscriptor<2> as the Subscriptor. The resulting class DAG is shown in Fig. 13.1.

Each of the four layers in this DAG provides a portion of the full behavior of the class at the bottom. At the top, ConcreteArrayShape<2> represents the shape of the array. For example, it represents the pair {3, 2} for a 3×2 matrix. Next, ConcreteColumnMajorSubscriptor<2> adds functions to compute the offset from an array origin to an element of an array stored in column-major format. This class depends only on the structure of the array and not on the type of the elements.

Third from the top comes ConcreteArray2d<Subscriptor, T> with Subscriptor equal to ConcreteColumnMajorSubscriptor<2>. This class adds storage for contiguous data and provides element access to the data. The relation between data storage and its access is computed by the offset() function in the subscriptor base class. Finally, at the bottom of the DAG comes ConcreteFortranArray2d<T>. This class has only constructors and assignment operators that determine the array object's flexibility. In this case, the size of the array is set when it is created.

A ConcreteFortranArray2d<double> object is sketched in Fig. 13.2. The array shape and its control are embedded inside the array class along with a pointer to the contiguous storage holding the elements. For ConcreteFortranArray2d<double>, the elements are on the heap and the remaining data—a pointer and two integers—are on the heap or stack according to the allocation of the array object itself.

We now look at the definitions of these four classes, showing the parameterized template code instead of the specific template classes: ndim rather than 2 appears and Subscriptor is used in place of ConcreteColumnMajorSubscriptor<2>. In other words, although we have in mind one particular template class, ConcreteFortranArray2d<T>, we show its class template as it appears in C++ code.

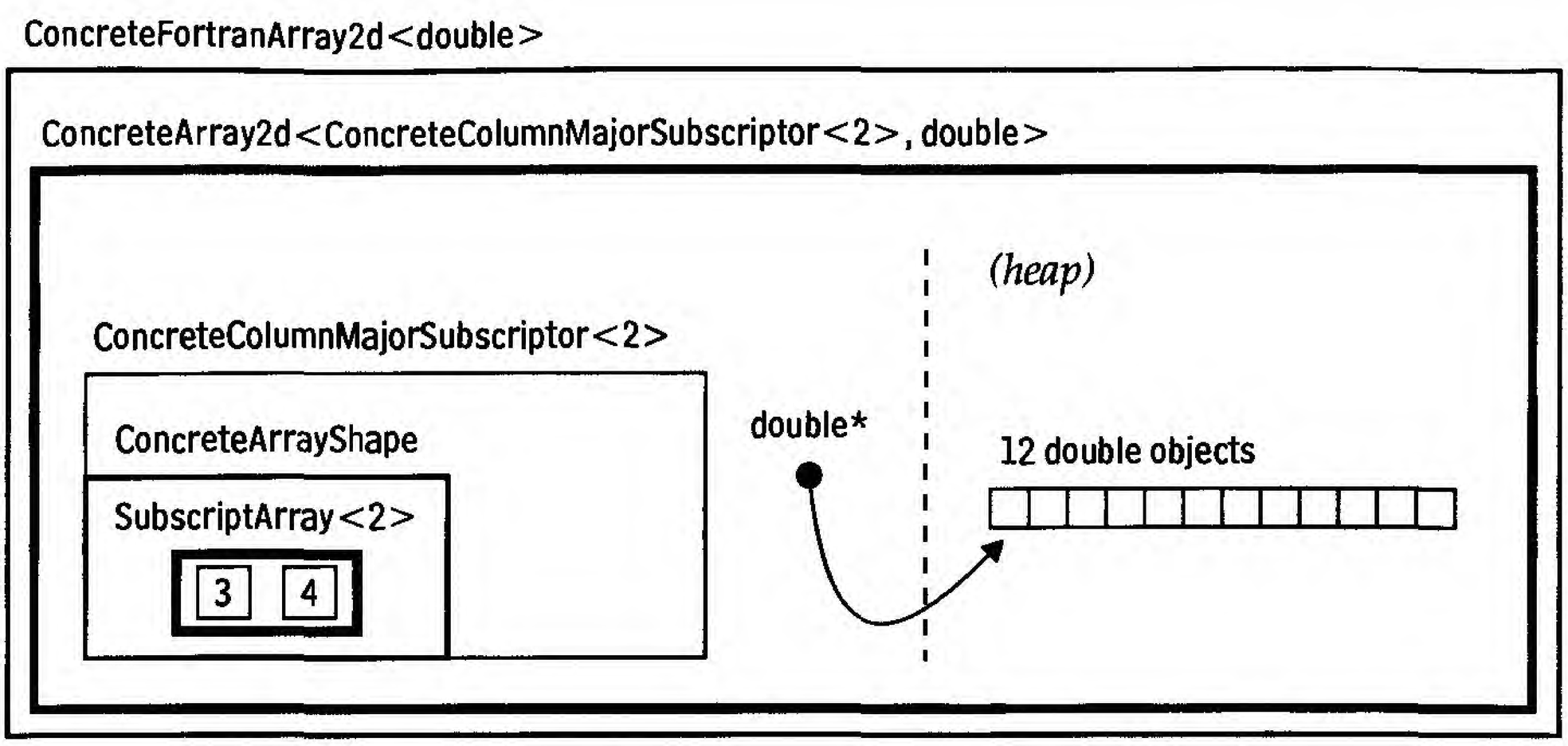


Figure 13.2 Sketch of a ConcreteFortranArray2d<double> object. The subscriptor is encapsulated in the array class, but the array shape is a public base class of the subscriptor. Bold lines mean encapsulation with no virtual function interface; thin lines mean public inheritance without encapsulation.

13.2.2 ConcreteArrayShape<ndim>

At the top of the concrete array system lies ConcreteArrayShape<ndim>, a representation of the shape of an array. The shape is held as a one-dimensional array of Subscripts, with member functions to access or set the appropriate information:

Array/ConcreteArrayShape.h

```
template<Dimension ndim>
class ConcreteArrayShape {
public:
    ConcreteArrayShape(const SubscriptArray<ndim>&);
    ConcreteArrayShape() {}    // Create uninitialized; call setShape to set.

    Dimension dim()                    const { return ndim;          }
    Subscript  shape(Dimension d) const { return the_shape(d); }
    Subscript  numElts()               const;
    void       setShape(const SubscriptArray<ndim>& a);
protected:
    SubscriptArray<ndim> the_shape;
};
```

The shape is held in a SubscriptArray<ndim> object (page 347), a rigid array of size ndim, fixed at compile time. This implementation of array shape requires ndim * sizeof(Subscript) words of storage (e.g., 8 bytes for ndim == 2 on a machine with 4-byte integers).

13.2.3 ConcreteColumnMajorSubscriptor<ndim>

Building directly on the representation for the shape are various *subscriptor* classes. These classes compute the offset for an element, given an array of subscripts. They also provide a connection to an appropriate projection subscriptor class of lower dimension. ConcreteFortranArray2d<T> makes use of the following subscriptor:

Array/ConcreteArrayShape.h

```
template<Dimension ndim>
class ConcreteColumnMajorSubscriptor :
    public ConcreteArrayShape<ndim> {
public:
    ConcreteColumnMajorSubscriptor(const SubscriptArray<ndim>& s);
    ConcreteColumnMajorSubscriptor() {}

    typedef ConcreteColumnMajorProjectionSubscriptor<ndim-1> ProjectionT;
    ProjectionT projectionSubscriptor(Dimension d, Subscript s) const;

    Subscript offset(const SubscriptArray<ndim>& s) const;
};
```

The first constructor (definition not shown) uses its argument to initialize the implementation base. The ProjectionT typedef gives a common name for the type of the subscriptor object appropriate for subscripting a projection of an array stored with the layout described by a ConcreteColumnMajorSubscriptor<ndim>. The projectionSubscriptor() function returns an instance of that type. This function is described in Section 13.4.1. Notice that the ProjectionT typedef refers to a class template with the dimensionality, ndim-1, computed by the compiler from the template parameter ndim.

The subscriptor for one-dimensional arrays is necessarily different, for there can be no projection subscriptor: The projection of a one-dimensional array is an element. This special case is handled by a template specialization that omits both the typedef and the projectionSubscriptor() member function:

Array/ConcreteArrayShape.h

```
class ConcreteColumnMajorSubscriptor<1> :
    public ConcreteArrayShape<1> {
public:
    ConcreteColumnMajorSubscriptor(const SubscriptArray<1>& s):
                                                    ConcreteArrayShape<1>(s) {}
    ConcreteColumnMajorSubscriptor() {}

    Subscript offset(const SubscriptArray<1>& s) const;
};
```

For all dimensionalities, the offset() function computes the integral offset from the beginning of the underlying element storage to the element given by the subscripts passed as arguments. Here is the function for our ConcreteFortranArray2d<T> example:

Array/ConcreteArrayShape.h

```
template<Dimension ndim>
inline
Subscript ConcreteColumnMajorSubscriptor<ndim>::
                                        offset(const SubscriptArray<ndim>& s) const {
    Dimension n = ndim;
    Subscript off = 0;
    while (n-- > 0) off = off * shape(n) + s(n);
    return off;
}
```

This template function works for arbitrary positive values of ndim, even 1: We need not define a template specialization for ConcreteColumnMajorSubscriptor<1>'s offset() member, even though it was declared in a class template specialization. On the other hand, if the performance of array subscripting is important (e.g., for small values of ndim that are used frequently), template specialization can be used to eliminate the loop in the template and improve the probability that the compiler will inline the function. See Exercise 13.1.

13.2.4 ConcreteArray2d<Subscriptor, T>

Moving down the DAG in Fig. 13.1, we come to the core of the concrete array system, ConcreteArray2d<Subscriptor, T>. This class is long, so we present it in several parts. As a sneak preview, the class contains a single member datum, a T* called datap, which points to the first element of a contiguous storage region managed by the class. In fact, manipulating this pointer is the class's primary role.

The first template parameter, Subscriptor, defines the shape and offset calculations for the array. It is used as a private base class with access declarations used to provide public access to the inherited functions:

Array/ConcreteArray2d.h

```
template<class Subscriptor, class T>
class ConcreteArray2d :
    private Subscriptor {
public:
    Subscriptor::dim;
    Subscriptor::shape;
    Subscriptor::numElts;
    Subscriptor::offset;
    Subscriptor::setShape;
```

The ConcreteColumnMajorSubscriptor<2> described in the preceding subsection is an appropriate Subscriptor for two-dimensional, FORTRAN-compatible arrays.

Next come typedef declarations for element type, subscriptor type, the types returned by non-const and const projection functions, and the types of the browser and iterator objects that can be used with these arrays:

Array/ConcreteArray2d.h

```
typedef T                                                   EltT;
typedef Subscriptor                                         SubscriptorT;

typedef ConcreteArrayProjection1d< Subscriptor, T >         ProjectionT;
typedef ConstConcreteArrayProjection1d< Subscriptor, T > ConstProjectionT;

typedef ConcreteArrayBrowser< ConcreteArray2d<Subscriptor, T> >BrowserType;
typedef ConcreteArrayIterator< ConcreteArray2d<Subscriptor, T> > IteratorType;

const T& operator()(Subscript s0, Subscript s1) const;
T&       operator()(Subscript s0, Subscript s1);

Subscriptor subscriptor() const { return *this; }

T const * firstDatum() const { return datap; }
T       * firstDatum()       { return datap; }
```

The projection types are described in Section 13.4.3 and the browser and iterator types in Section 13.7.3. The function call operators, which provide subscript access, are defined on page 377. We also provide access to the subscriptor object and to the origin of storage for the elements. The latter functions are dangerous in that their callers might hold onto the pointer returned even after the array is destroyed. Such a pointer is a dangling reference, as described in Section 7.6. Such dangers always accompany code designed for minimum overhead and compatibility with other languages: Passing an array to a FORTRAN function means passing a pointer to the first element.

Projections are provided by the project() functions, with various shorthands provided by inline functions that call project():

Array/ConcreteArray2d.h

```
ConstProjectionT project(Subscript s, Dimension d) const;
ProjectionT      project(Subscript s, Dimension d);

ConstProjectionT operator[ ](Subscript s) const { return project(s, 0); }
ProjectionT      operator[ ](Subscript s)       { return project(s, 0); }

ConstProjectionT row(Subscript i)       const { return project(i, 0); }
ProjectionT      row(Subscript i)             { return project(i, 0); }

ConstProjectionT column(Subscript i)    const { return project(i, 1); }
ProjectionT      column(Subscript i)          { return project(i, 1); }
```

The project() functions and the objects they return are described in Section 13.4.3.

Finally, we have conversion operators, assignment operators, constructors, and some other functions that provide services to derived classes:

Array/ConcreteArray2d.h

```
    operator ConcreteArray2dConstRef<Subscriptor, T>() const;
    operator ConcreteArray2dRef<Subscriptor, T>();

    ConcreteArray2d<Subscriptor, T>&
                        operator=(const ConcreteArray2d<Subscriptor, T>& rhs);
    ConcreteArray2d<Subscriptor, T>&
                        operator=(const ConcreteArray2dConstRef<Subscriptor, T>& rhs);
    ConcreteArray2d<Subscriptor, T>&
                        operator=(const T& rhs);
protected:
    ConcreteArray2d(const Subscriptor& s, T* p) : Subscriptor(s), datap(p) {}
    void reshapeOnHeap(const SubscriptArray<2>& s);
    void setSizeOnHeap(Subscript n);
protected:
    T* datap;
private:
    ConcreteArray2d(const ConcreteArray2d<Subscriptor,T>&);
};
```

The `ConcreteArray2dConstRef<Subscriptor, T>` and `ConcreteArray2dRef<Subscriptor, T>` classes are explained in Section 13.3; more specifically, the roles and implementations of the conversion and assignment operators are explained in Section 13.3.3.

Since this class has a built-in pointer data member, we know from the advice given in Section 7.6 that there should be a copy constructor. However, in our design, classes derived from this class handle memory allocation. Therefore, we declare a `private` copy constructor; this prevents C++ from generating a copy constructor but prohibits others from using the one we declare. In fact, we don't even bother to supply a definition for the copy constructor since this class doesn't call it and no client can call it. A derived array class that wants to allow its instances to be copied must supply appropriate copy constructors.

The only accessible constructor is `protected`: It is meant to be called only by derived classes like `ConcreteFortranArray2d<T>`. This base class constructor requires a subscriptor and a pointer to allocated memory; it uses these to initialize its subscriptor base subobject and its built-in pointer, respectively, as shown in the constructor definition in the class body.

The functions that are not defined in the body of the class include functions for allocation, access, conversion, assignment, and projection. We show the allocation and access functions here and defer discussion of the conversion, assignment (Section 13.3.3), and projection functions (Section 13.4.3).

The allocation functions setSizeOnHeap() and reshapeOnHeap() are not called by ConcreteArray2d<T> but are provided here for use by derived classes:

Array/ConcreteArray2d.c

```
template<class Subscriptor, class T>
void ConcreteArray2d<Subscriptor, T>::setSizeOnHeap(Subscript n) {
    if (n < 0) throw ArrayErr::NegativeSize();
    delete[] datap;
    datap = new T[n];
}

template<class Subscriptor, class T>
void ConcreteArray2d<Subscriptor, T>::reshapeOnHeap(const SubscriptArray<2>& s) {
    setShape(s);
    setSizeOnHeap(numElts());
}
```

For example, ConcreteFortranArray2d<T> provides reshape() for explicit shape changes and uses reshapeOnHeap() in its definition. Using these functions, an elastic array that reshapes automatically to accommodate the size of the array on the right-hand side of assignments can be written (see Exercise 13.3). Putting setSizeOnHeap() and reshapeOnHeap() here, but making them protected, allows derived classes to inherit a common implementation of heap allocation while leaving open the possibility of alternative storage for the array elements.

That leaves the subscripting operators:

Array/ConcreteArray2d.h

```
template<class Subscriptor, class T>
inline
const T& ConcreteArray2d<Subscriptor, T>::operator()(Subscript s0, Subscript s1) const {
    return firstDatum()[offset(SubscriptArray<2>(s0, s1))];
}

template<class Subscriptor, class T>
inline
T& ConcreteArray2d<Subscriptor, T>::operator()(Subscript s0, Subscript s1) {
    return firstDatum()[offset(SubscriptArray<2>(s0, s1))];
}
```

These just combine the offset() calculation with the element pointer from firstDatum(). The combination uses square brackets on the pointer, supplying an integer offset. C++ treats the pointer as the first element of an array and steps forward from it by the offset times the size of T objects. Different Subscriptor template parameters will cause different calculations of the offset() using the same function call signature.

13.2.5 ConcreteFortranArray2d<T>

At the bottom of the class DAG in Fig. 13.1 lives ConcreteFortranArray2d<T>:

Array/ConcreteFortranArray2d.h

```
template<class T>
class ConcreteFortranArray2d :
    public ConcreteArray2d<ConcreteColumnMajorSubscriptor<2>, T > {
public:
    ConcreteFortranArray2d(Subscript s0, Subscript s1);
    ConcreteFortranArray2d(const ConcreteFortranArray2d<T>& p);
ConcreteFortranArray2d(
    const ConcreteArray2dConstRef<ConcreteColumnMajorSubscriptor<2>, T>&);
ConcreteFortranArray2d(
    const ConcreteArray2dConstRef<
                        ConcreteColumnMajorSubscriptor<3>::ProjectionT, T>&);
ConcreteFortranArray2d(const ConstArray2d<T>&);

~ConcreteFortranArray2d();

ConcreteFortranArray2d<T>& operator=(const ConcreteFortranArray2d<T>& rhs);
ConcreteFortranArray2d<T>& operator=(
    const ConcreteArray2dConstRef<SubscriptorT, T>& rhs);
    ConcreteFortranArray2d<T>& operator=(const T& rhs);

    void reshape(const SubscriptArray<2>& s) { reshapeOnHeap(s); }
};
```

There are no member data and only creation-related member functions: This class exists to set the subscriptor to column major and create the element storage on the heap in its constructors and assignment operators. This class fixes both the storage allocation strategy—runtime determination of size, which is not changed by assignment—and the subscripting strategy—which is column-major storage compatible with FORTRAN, but with zero subscript origin like C and C++ built-in arrays.

As examples, we show two constructors. The first one takes the number of rows and columns and uses the inherited member function setSizeOnHeap() (page 377) to allocate the appropriate memory:

Array/ConcreteFortranArray2d.c

```
template<class T>
ConcreteFortranArray2d<T>::ConcreteFortranArray2d(Subscript s0, Subscript s1) :
    ConcreteArray2d<SubscriptorT, T>(SubscriptArray<2>(s0, s1), 0) {
    setSizeOnHeap(s0 * s1);
}
```

The call to the base class constructor creates a temporary object by calling the SubscriptArray<2> constructor explicitly.

The copy constructor is similar, except that the sizes are taken from the original array instead of being passed in as arguments, and the elements are copied using a function to be defined on page 383:

Array/ConcreteFortranArray2d.c

```
template<class T>
ConcreteFortranArray2d<T>::
ConcreteFortranArray2d(const ConcreteFortranArray2d<T>& a) :
    ConcreteArray2d<SubscriptorT, T>(SubscriptArray<2>(a.shape(0), a.shape(1)), 0) {
    setSizeOnHeap(a.numElts());
    concreteCopy(*this, a);
}
```

The concreteCopy() function is used for element copies in all of the arrays and array projections.

Using the classes described so far, we can create arrays stored like FORTRAN arrays and access their elements with subscripts. Projections are also vital to effective array classes. We have to build up a bit more mechanism before we are ready to tackle projections themselves.

13.3 Concrete Array References

A projection is a reduced-dimensionality view of an array. Importantly, a projection is not a copy of a portion of an array, but rather a reference to a portion of an array: A projection refers to the elements of an existing array and is not itself an array. Nevertheless, as with a built-in reference, we want a reference to an array to behave like an array. Unlike a built-in reference to an array, which refers to the entire array, we want projection to be a reference to a selected subset of an array.

Projections are not the only kind of references to arrays that would be useful. For example, we might want to write a reference to the diagonal elements of an array (those for which all subscripts are equal). Moreover, the read-only character of a const array must be maintained for both projections and other references to the array. For these reasons, we implement projections in two steps, first defining array reference objects and then using them to implement projections. We tackle array references here and projections in Section 13.4.

13.3.1 Behavior

To understand the behavior we need from array reference objects, let's look at the behavior we get with built-in reference objects. Consider the following simple example:

ch13/tCollect.C

```
int i = 5;                  // Object, value 5.
int &ir = i;                // non-const reference to i.
const int& cir = i;         // const reference to i.
const int& cir1 = ir;       // const reference to i.
int& ir1 = cir;             // WRONG: const int& can't be converted to int&
ir = 6;                     // Alter object i
```

Both ir and cir are (built-in) references, and they both refer to the same object. That object can be modified through the non-const reference ir, but not through the const reference cir. Since the const reference is more restrictive than the non-const reference, a const reference (cir1) can be initialized from a non-const reference (ir), but not vice versa.

When one reference is initialized from another, the object referred to is not copied. Once initialized, a reference always refers to the same object; the reference itself cannot be modified. Assigning to a (non-const) reference means assigning to the referenced object.

We want our array reference objects to behave like built-in references with the added ability to refer to portions of an array. This goal leads immediately to several design decisions:

- Array reference objects should behave like arrays: subscripting, projection, shape query, etc., should all work.
- Once initialized, an array reference object should not be modifiable. This implies that all of its member functions should be const.
- Copying an array reference object does not copy the object to which it refers.
- Assigning to an array reference object assigns to the array elements to which the reference refers.
- Two kinds of array reference objects are needed: one that behaves like a const reference and one that behaves like a non-const reference. It should be possible to convert the reference object that acts like a non-const reference to one that behaves like a const reference, but not vice versa.

13.3.2 Array Reference Classes

These decisions lead us to define two kinds of array reference objects. The first one, shown here in its two-dimensional version, behaves like a const reference to an array:

Array/ConcreteArray2d.h

```
template<class Subscriptor, class T>
class ConcreteArray2dConstRef :
    private Subscriptor {
```

```
public:
    Subscriptor::dim;
    Subscriptor::shape;
    Subscriptor::numElts;
    Subscriptor::offset;

    typedef T                                                    EltT;
    typedef Subscriptor                                          SubscriptorT;
    typedef ConstConcreteArrayProjection1d<Subscriptor, T>       ConstProjectionT;

    typedef ArrayBrowser2d< ConcreteArray2dConstRef<Subscriptor, T> > BrowserType;

    const T& operator()(Subscript s0, Subscript s1) const {
        return firstDatum()[offset(SubscriptArray<2>(s0, s1))];
    }

    ConstProjectionT project(Subscript, Dimension) const;

    ConstProjectionT operator[](Subscript s) const { return project(s, 0); }
    ConstProjectionT row(Subscript s)        const { return project(s, 0); }
    ConstProjectionT column(Subscript s)     const { return project(s, 1); }

    const T* firstDatum() const { return datap; }

    Subscriptor subscriptor() const { return *this; }
protected:
    ConcreteArray2dConstRef(Subscriptor s, const T* p) : Subscriptor(s), datap(p) {}
    friend ConcreteArray2dRef<Subscriptor, T>;
    friend ConcreteArray2d<Subscriptor, T>;
private:
    void operator=(const ConcreteArray2dConstRef<Subscriptor, T>&);    // Prohibit
private:
    const T* const datap;
};
```

This class is remarkably similar to the ConcreteArray2d<Subscriptor, T> class shown in Section 13.2.4. The major differences are that all the member functions are const and that the pointer to the array elements both points to const elements and is const itself. In other words, neither the reference object nor the elements to which it refers can be modified.

Clients are prohibited from assigning to one of these reference objects since neither the reference object nor the array to which it refers can be altered. The constructor is protected because this class is intended to be used to implement derived classes that refer to particular portions of an array. The friend declarations exist to allow these reference objects to be made from both non-const-like array reference objects and array objects via conversion operators that call the protected constructor. The definition of the project() member is shown on page 393; all other functions are defined in the body of the class.

The second array reference class behaves like a non-const reference to an array:

Array/ConcreteArray2d.h

```
template<class Subscriptor, class T>
class ConcreteArray2dRef :
      private Subscriptor {
public:
      Subscriptor::dim;
      Subscriptor::shape;
      Subscriptor::numElts;
      Subscriptor::offset;

      typedef T              EltT;
      typedef Subscriptor SubscriptorT;

      typedef ConstConcreteArrayProjection1d<Subscriptor, T> ConstProjectionT;
      typedef ConcreteArrayProjection1d<Subscriptor, T>       ProjectionT;

      typedef ArrayBrowser2d< ConcreteArray2dRef<Subscriptor, T> > BrowserType;
      typedef ArrayIterator2d< ConcreteArray2dRef<Subscriptor, T> > IteratorType;

      T&          operator()(Subscript s0, Subscript s1) const;
      Subscriptor subscriptor()                          const { return *this; }
      T*          firstDatum()                           const { return datap; }

      ProjectionT project(Subscript, Dimension) const;

      ProjectionT operator[](Subscript s) const { return project(s, 0); }
      ProjectionT row(Subscript s)        const { return project(s, 0); }
      ProjectionT column(Subscript s)     const { return project(s, 1); }

      operator ConcreteArray2dConstRef<Subscriptor, T>() const;
```

```
    const ConcreteArray2dRef<Subscriptor, T>& operator=(
        const ConcreteArray2dConstRef<Subscriptor, T>& rhs) const;
    const ConcreteArray2dRef<Subscriptor, T>& operator=(
        const ConcreteArray2dRef<Subscriptor, T>& rhs) const;
    const ConcreteArray2dRef<Subscriptor, T>& operator=(const T& rhs) const;
protected:
    ConcreteArray2dRef(Subscriptor s, T* p) : Subscriptor(s), datap(p) {}
    friend ConcreteArray2d<Subscriptor, T>;
private:
    T* const datap;
};
```

In this class, the pointer to the elements is const and therefore can't be altered, but the elements can be altered through the pointer.

Unlike our const-like array reference, this array reference provides assignment operators; they alter the elements to which the array reference object refers, not the array reference object itself. Therefore the assignment operator member functions are all const member functions and return const references to the left-hand side. The scalar assignment operator is implemented recursively by assigning to all of the one-dimensional subarrays obtained by projection:

Array/ConcreteArray2d.c

```
template<class Subscriptor, class T>
const ConcreteArray2dRef<Subscriptor, T>&
ConcreteArray2dRef<Subscriptor, T>::operator=(const T& rhs) const {
    Subscript n = shape(0);
    while (n-- > 0) (*this)[n] = rhs;
    return *this;
}
```

The other two assignment operators are implemented using a concreteCopy() function template. This global function template copies the elements of its second argument into the elements of its first element:

Array/ConcreteArray2d.c

```
template<class ArrayShape, class AnotherShape, class T>
void concreteCopy(
        const ConcreteArray2dRef<ArrayShape, T>& lhs,
        const ConcreteArray2dConstRef<AnotherShape, T>& rhs
    ) {
    Subscript n = lhs.shape(0);
    if (n != rhs.shape(0)) throw ArrayErr::Shape();
    while (n-- > 0) concreteCopy(lhs[n], rhs[n]);
}
```

This function is written in terms of projections that yield lower dimensional array reference objects. The lowest dimension concreteCopy() moves elements:

Array/ConcreteArray1d.c

```
template<class ArrayShape, class AnotherShape, class T>
void concreteCopy(
        ConcreteArray1dRef<ArrayShape, T> lhs,
        ConcreteArray1dConstRef<AnotherShape, T> rhs
    ) {
    Subscript n = lhs.shape(0);
    if (n != rhs.shape(0)) throw ArrayErr::Shape();
    while (n-- > 0) lhs[n] = rhs[n];
}
```

Template specialization could be used to supply faster code for specific cases.

Here then are the two assignment operators for array references:

Array/ConcreteArray2d.c

```
template<class Subscriptor, class T>
const ConcreteArray2dRef<Subscriptor, T>&
ConcreteArray2dRef<Subscriptor, T>::operator=(
        const ConcreteArray2dConstRef<Subscriptor, T>& rhs
    ) const {
    concreteCopy(*this, rhs);
    return *this;
}

template<class Subscriptor, class T>
const ConcreteArray2dRef<Subscriptor, T>&
ConcreteArray2dRef<Subscriptor, T>::operator=(
        const ConcreteArray2dRef<Subscriptor, T>& rhs
    ) const {
    // Convert rhs reference to const reference and use its assignment
    const ConcreteArray2dConstRef<Subscriptor, T> rhsa = rhs;
    return *this = rhs;
}
```

Conversion from a non-const-like array reference object to a const-like array reference object is also provided; it is implemented by calling the appropriate constructor:

Array/ConcreteArray2d.h

```
template<class Subscriptor, class T>
inline
ConcreteArray2dRef<Subscriptor, T>::
operator ConcreteArray2dConstRef<Subscriptor, T>() const {
    return ConcreteArray2dConstRef<Subscriptor, T>(*this, datap);
}
```

Here the friend declaration in the ConcreteArray2dConstRef<Subscriptor, T> template allows access to the constructor.

13.3.3 Conversion and Assignment of Arrays

Now that we understand what array reference objects are and how they are implemented, let's look at how they are connected to our arrays. Section 13.4 will connect array references to array projections.

Array reference objects are only useful if they can be made from various kinds of arrays. The constructors for the array reference objects are protected, so a general client can't create one. We'll see in Section 13.4 that array projection objects are derived from array references and thus create array reference objects as base subobjects. Array reference objects can also be created by conversion from array objects, via the conversion operators declared in ConcreteArray2d<Subscriptor, T> on page 376. Access was granted to ConcreteArray2d<Subscriptor, T> by the friend declaration on page 380.

The conversion implementation is similar to what one might do in C to refer to an array. The array reference object simply contains a copy of the array's subscriptor and a copy of its pointer to its elements:

Array/ConcreteArray2d.h

```
template<class Subscriptor, class T>
inline
ConcreteArray2d<Subscriptor, T>::
operator ConcreteArray2dConstRef<Subscriptor, T>() const {
    return ConcreteArray2dConstRef<Subscriptor, T>(subscriptor(), datap);
}

template<class Subscriptor, class T>
inline
ConcreteArray2d<Subscriptor, T>::operator ConcreteArray2dRef<Subscriptor, T>() {
    return ConcreteArray2dRef<Subscriptor, T>(subscriptor(), datap);
}
```

Note that a const-like array reference is made for a const array and a non-const-like array reference is made for a non-const array. These distinctions, and the packaging of the pointer and subscripts, are unlike the C equivalent.

The allowable conversions among concrete arrays, concrete array reference objects, and concrete projection objects are shown in Fig. 13.3. For example, here is a conversion from an array to an array reference object:

ch13/tCollect.C

```
typedef ConcreteFortranArray2d<double> Array;    // Abbreviation
Array a(3, 4);
ConcreteArray2dRef<Array::SubscriptorT, Array::EltT> ar = a;
ar = 10.0;
```

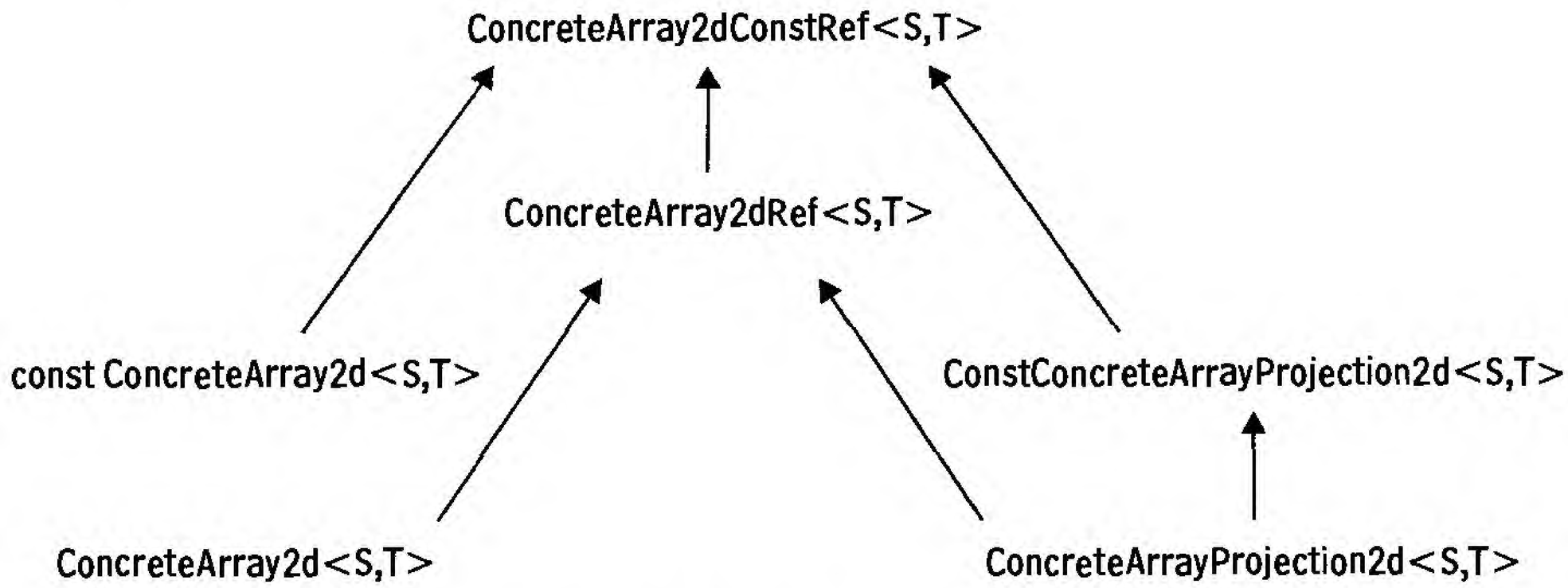


Figure 13.3 Allowable Conversions among Concrete Arrays, Concrete Array Reference Objects, and Concrete Projection Objects. Arrows indicate that an instance of the class named at the tail can be converted to an instance of the class named at the arrowhead; arrows do not indicate derivation.

The array reference object ar refers to the elements of the array a; the reference object is illustrated in Fig. 13.4.

The assignment operators declared in ConcreteArray2d<Subscriptor, T> are all implemented using array reference objects:

Array/ConcreteArray2d.h

```
template<class Subscriptor, class T>
inline ConcreteArray2d<Subscriptor, T>&
ConcreteArray2d<Subscriptor, T>::
operator=(const ConcreteArray2dConstRef<Subscriptor, T>& rhs) {
    ConcreteArray2dRef<Subscriptor, T>(*this) = rhs;
    return *this;
}

template<class Subscriptor, class T>
inline ConcreteArray2d<Subscriptor, T>&
ConcreteArray2d<Subscriptor, T>::
operator=(const ConcreteArray2d<Subscriptor, T>& rhs) {
    ConcreteArray2dRef<Subscriptor, T>(*this) = rhs;
    return *this;
}

template<class Subscriptor, class T>
inline ConcreteArray2d<Subscriptor, T>&
ConcreteArray2d<Subscriptor, T>::operator=(const T& rhs) {
    ConcreteArray2dRef<Subscriptor, T>(*this) = rhs;
    return *this;
}
```

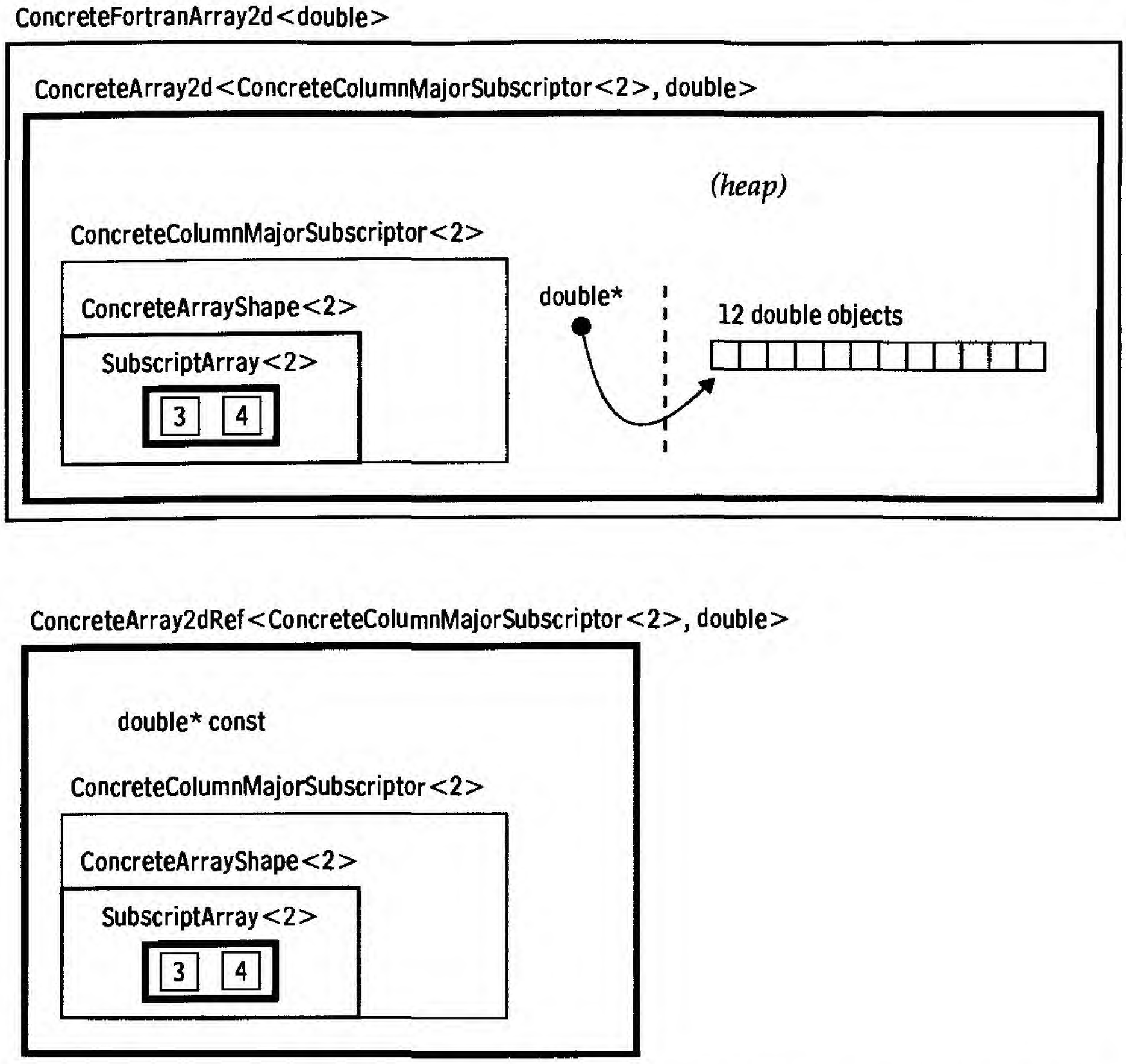


Figure 13.4 Sketch of an Array Reference Object That Refers to an Entire Array. Bold lines mean encapsulation with no virtual function interface; thin lines mean public inheritance without encapsulation.

These simply convert the left-hand side to an array reference object and let the array reference assignment operators do the work. The cost of creating the array reference object is small: copying two subscripts and a pointer. Except for arrays with only a few elements, the conversion cost is negligible; of course, the assignment operators could be implemented directly if measurements dictate doing so.

13.3.4 Dangling References

The array reference objects we have developed behave much like built-in references. Unfortunately, this behavior includes the dangling reference problem, as described in Section 7.6. For example, the function dangle() returns a dangling reference in the following code:

```
typedef ConcreteFortranArray2d<double>                          Array;  // Abbreviations
typedef ConcreteArray2dRef<Array::SubscriptorT, Array::EltT> Ref;
```
ch13/tCollect.C

```
Ref dangle() {
    Array a(3, 4);
    return Ref(a);    // Creates dangling reference
}
```

The returned array reference object refers to elements that will be deleted when the function returns, leaving a dangling reference.

Correct use of references (built in or programmer defined) requires that the lifetime of the referenced object exceed the lifetime of the reference. For concrete arrays, our solution is to hope for the best. This is similar to what C++ does for pointers and built-in references. Moreover, this strategy is the only one currently available that avoids runtime overhead.

In practice the problem is not severe. Most array reference objects are created as temporaries in expressions containing the array being referred to, or as arguments to functions that will be called from functions containing the scope of the array object. Problems occur when we try to use projections as member data and forget to synchronize the new class with the scope of the array.

13.4 Concrete Array Projections

We now have the infrastructure for implementing array projections: An array projection object is an array reference object having a subscriptor that selects only a portion of the array's elements. We illustrate array projections with the running example from Section 13.2, `ConcreteFortranArray2d<T>`.

Projection on a two-dimensional array yields a one-dimensional projection. Projection of a const array object must yield a const projection; projection of a non-const array object must not be a const projection. Thus in this section we build two concrete projection classes, `ConcreteArrayProjection1d<Subscriptor, T>` and `ConstConcreteArrayProjection1d<Subscriptor, T>`.

Figure 13.5 diagrams the class DAG for non-const projections of non-const `ConcreteFortranArray2d<T>`. Compare this diagram to Fig. 13.1. Notice that `ConcreteArray1dRef<Subscriptor, T>` is a base class for `ConcreteArrayProjection1d<Subscriptor,T>` in the same way that `ConcreteArray2d<Subscriptor, T>` is the base for `ConcreteFortranArray2d<T>`. These base classes serve similar roles in their respective class DAGs, holding pointers to elements and providing subscript computations but not providing constructors.

We work down this DAG as we did for the `ConcreteFortranArray2d<T>` DAG in Section 13.2. The first class, `ConcreteArrayShape<ndim>`, is reused from page 372.

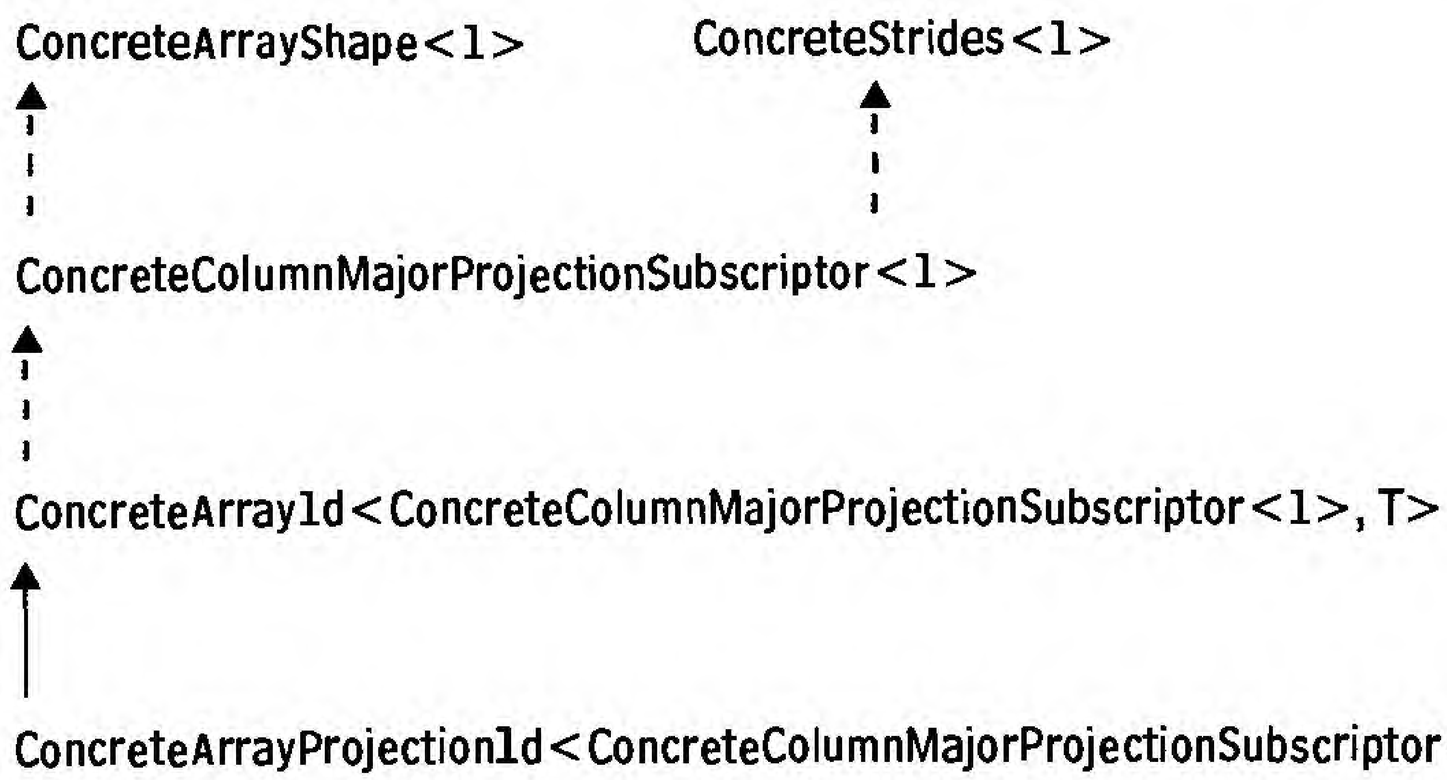


Figure 13.5 Class DAG for Projections of ConcreteFortranArray2d<T> Objects.

Thus we begin with the projection subscriptor analogous to ConcreteColumnMajorSubscriptor<ndim> on page 373 and its private base class ConcreteStrides<ndim>.

13.4.1 ConcreteColumnMajorProjectionSubscriptor<ndim>

For our FORTRAN-compatible arrays, the array subscriptor can compute the offset of an element from the shape and subscripts alone. The projection subscriptor, on the other hand, must skip over the elements in the dimension projected away: Computing the offset for a projection requires knowing the stride (see Section 13.2.1) in each dimension of the projection. Our projection subscriptors represent the strides with the following class:

Array/ConcreteArrayShape.h

```
template<Dimension ndim>
class ConcreteStrides {
public:
    ConcreteStrides(const SubscriptArray<ndim>& s) : the_strides(s) {}

    Subscript stride(Dimension d)                          const;
    Subscript offset(const SubscriptArray<ndim>& s) const;
protected:
    ConcreteStrides() {}
    SubscriptArray<ndim> the_strides;
};
```

The strides are different for different storage layouts, but the computation of the offset() from strides is layout independent:

Array/ConcreteArrayShape.h

```
template<Dimension ndim>
inline
Subscript ConcreteStrides<ndim>::offset(const SubscriptArray<ndim>& s) const {
    Subscript off = 0;
    Dimension d = ndim;
    while (d-- > 0) off += stride(d) * s(d);
    return off;
}
```

The projection subscriptor glues together the shape implemented in ConcreteArrayShape<ndim> with this offset() computation in ConcreteStrides<ndim>:

Array/ConcreteArrayShape.h

```
template<Dimension ndim>
class ConcreteColumnMajorProjectionSubscriptor :
    private ConcreteArrayShape<ndim>,
    private ConcreteStrides<ndim> {
public:
    ConcreteColumnMajorProjectionSubscriptor(
        const SubscriptArray<ndim>& shape,
        const SubscriptArray<ndim>& strides
    );
    ConcreteColumnMajorProjectionSubscriptor() {}

    typedef ConcreteColumnMajorProjectionSubscriptor<ndim-1> ProjectionT;

    ConcreteArrayShape<ndim>::dim;
    ConcreteArrayShape<ndim>::shape;
    ConcreteArrayShape<ndim>::numElts;
    ConcreteArrayShape<ndim>::setShape;
    ConcreteStrides<ndim>::offset;

    ProjectionT projectionSubscriptor(Dimension d, Subscript s) const;
};
```

The glue is inheritance. We add a typedef referring to the projection subscriptor type for this projection subscriptor (i.e., the subscriptor for projections of projections) and a function for computing a projection of this projection. Its definition appears on page 394.

Notice that the typedef gives a name to the same class, with the dimensionality template parameter reduced by one. In a two-dimensional projection subscriptor, the typedef refers to a one-dimensional projection subscriptor; in a one-dimensional projection subscriptor, the typedef refers to a zero-dimensional

projection subscriptor, and so on. Clearly we have a problem: This template recurses on dimensionality and we need some way to stop the recursion. The solution is to use a template specialization for the one-dimensional case:

Array/ConcreteArrayShape.h

```
class ConcreteColumnMajorProjectionSubscriptor<1> :
    private ConcreteArrayShape<1>,
    private ConcreteStrides<1> {
public:
    ConcreteColumnMajorProjectionSubscriptor(
        const SubscriptArray<1>& shape,
        const SubscriptArray<1>& strides
    );
    ConcreteColumnMajorProjectionSubscriptor() {}

    ConcreteArrayShape<1>::dim;
    ConcreteArrayShape<1>::shape;
    ConcreteArrayShape<1>::numElts;
    ConcreteArrayShape<1>::setShape;
    ConcreteStrides<1>::offset;
};
```

This projection subscriptor omits the typedef and the projectionSubscriptor() member function, stopping the recursion.

- Use template specialization to terminate template recursion.

13.4.2 ConcreteArrayProjection1d<Subscriptor, T>

In the same way that ConcreteFortranArray2d<T> set the Subscriptor and added creation routines onto its ConcreteArray2d<Subscriptor, T> base class, a ConcreteArrayProjection1d<Subscriptor, T> sets the Subscriptor and adds creation routines onto its ConcreteArray1dRef<Subscriptor,T> base class (not shown; see Exercise 13.7). As we see here, the Subscriptor for projections is set symbolically in terms of the Subscriptor for the array being projected:

Array/ConcreteArrayProjection1d.h

```
template<class Subscriptor, class T>
class ConcreteArrayProjection1d :
    public ConcreteArray1dRef<Subscriptor::ProjectionT, T> {
public:
    typedef Subscriptor::ProjectionT ProjectionSubscriptor;
    ConcreteArrayProjection1d(const ProjectionSubscriptor& s, T* t);
```

```
        operator ConstConcreteArrayProjection1d<Subscriptor, T>() const;

        ConcreteArrayProjection1d<Subscriptor, T>&
            operator=(ConcreteArray1dConstRef<ProjectionSubscriptor, T> rhs);
        ConcreteArrayProjection1d<Subscriptor, T>&
            operator=(const T& rhs);
    protected:
        // Copying a projection does not copy the underlying array elements. It only duplicates
        // the reference to the underlying array. We make the copy constructor protected to avoid
        // accidental copies by client code.
        ConcreteArrayProjection1d(const ConcreteArrayProjection1d<Subscriptor, T>& proj);
};
```

The symbolic connection between projection subscriptors and array subscriptors is accomplished through template parameters alone.

Let's look at this connection in more detail. A projection object will be created with its Subscriptor parameter equal to the array's Subscriptor. Thus a projection of a ConcreteFortranArray2d<T> array (page 375) will be a ConcreteArrayProjection1d<ConcreteColumnMajorSubscriptor<2>, T>. The base class for this object will be ConcreteArray1dRef<ConcreteColumnMajorProjectionSubscriptor<1>,T>. By this means, we connect column-major subscript calculations for the array to compatible subscript calculations for the projections of the array. The subscriptor for the projection is a projection subscriptor, the specific projection subscriptor given by the ProjectionT typedef in the array subscriptor described in Section 13.2.3.

13.4.3 How Concrete Projections Work

Finally we put all of this projection work together by following the function calls made by executing the last statement in the following example:

ch13/tCollect.C

```
ConcreteFortranArray2d<double> a(3, 4);
a[1] = 1.0;
```

The first statement creates an array of 12 double values accessed as a two-dimensional array by column-major subscripting. The second statement sets the value one into row one. This example matches the illustration of the 3×4 array in Fig. 13.4.

The execution of the statement a[1]=1.0 has two parts. The first part is a series of function calls beginning with a[1] and ending with a temporary object containing a pointer and some indices that reference the first row of a. The second part calls the assignment operator on the temporary to load 1.0 into each element of the row. Now let's look at the details.

The square bracket operator for ConcreteFortranArray2d<double> is inherited from a class created from the ConcreteArray2d<Subscriptor,T> class template (page 375), with Subscriptor equal to ConcreteColumnMajorArraySubscriptor<2> and T equal to double. The operator expands inline to call project(1,0), the fundamental projection function, also inherited from the ConcreteArray2d<Subscriptor, T> class.

This project() function creates a temporary array reference object of type ConcreteArray2dRef<Subscriptor, T> from the array, and it then calls project() on the reference:

Array/ConcreteArray2d.h

```
template<class Subscriptor, class T>
inline
ConcreteArray2d<Subscriptor, T>::ProjectionT
ConcreteArray2d<Subscriptor, T>::project(Subscript s, Dimension d) {
    return ConcreteArray2dRef<Subscriptor, T>(*this).project(s, d);
}
```

We are sharing code here: The projections of an array and the projections of array reference objects are the same, so we do the work in one place, in the array reference class.

The array reference project() function creates a projection object, a ConcreteArrayProjection1d<Subscriptor, T>, passing it a subscriptor and a pointer to elements:

Array/ConcreteArray2d.c

```
template<class Subscriptor, class T>
ConcreteArrayProjection1d< Subscriptor, T >
ConcreteArray2dRef<Subscriptor, T>::project(Subscript s, Dimension d) const {
    SubscriptArray<2> pjs(0,0);
    pjs(d) = s;
    return ConcreteArrayProjection1d< Subscriptor, T >(
        projectionSubscriptor(d, s),
        firstDatum()+offset(pjs)
    );
}
```

We describe the projectionSubscriptor() in the next paragraph. The element pointer is biased to get to the first element of the projection. In this example, the Subscriptor is ConcreteColumnMajorSubscriptor<2>, the Dimension d is 0, and the Subscript s is 1. The offset() function is called with (1,0), giving an offset of 1 to be added to the pointer returned by firstDatum().

The subscriptor for the projection is returned by projectionSubscriptor(), a member of ConcreteColumnMajorSubscriptor<ndim>; it can compute the strides—the offset between projected elements—using the array offset() function:

Array/ConcreteArrayShape.c

```
template<Dimension ndim>
ConcreteColumnMajorProjectionSubscriptor<ndim-1>
ConcreteColumnMajorSubscriptor<ndim>::
projectionSubscriptor(Dimension proj_dim, Subscript) const {
    SubscriptArray<ndim-1> proj_shape;      // Shape of projection
    SubscriptArray<ndim-1> proj_strides;    // Strides of projection
    SubscriptArray<ndim> step;              // Holds subscripts into array

    Dimension j = 0;      // Dimension in projection
    Dimension k = 0;      // Corresponding dimension in array
    step = 0;             // Will be set to take one "step" in dimension k
    while (j < ndim-1) {
        if (j == proj_dim) ++k;  // Skip over projected dimension
        proj_shape(j) = shape(k); // Proj shape is array shape without dimension proj_dim

                                  // Compute offset by taking a step in dimension k
        step(k) = 1;
        proj_strides(j) = offset(step);
        step(k) = 0;

        j++;
        k++;
    }

    return ConcreteColumnMajorProjectionSubscriptor<ndim-1>(proj_shape, proj_strides);
}
```

The second argument is not needed for strided arrays; omitting the name of an unused argument indicates that you intended not to use the argument, thus suppressing warnings the compiler might otherwise have given. See page 562 for a case in which the second argument is needed.

§8.3

The constructor called in the return from the projectionSubscriptor() function loads the shape and strides into base class member data:

Array/ConcreteArrayShape.c

```
template<Dimension ndim>
ConcreteColumnMajorProjectionSubscriptor<ndim>::
ConcreteColumnMajorProjectionSubscriptor(
        const SubscriptArray<ndim>& the_shape,
        const SubscriptArray<ndim>& the_strides
    ):
    ConcreteArrayShape<ndim>(the_shape),
    ConcreteStrides<ndim>(the_strides) {
}
```

When this constructor returns, projectionSubscriptor() returns a projection subscriptor object.

This brings us back to the construction of the temporary ConcreteArrayProjection1d<Subscriptor,T> object to be returned by the project() function on page 393. That function passes the subscriptor object just created and a pointer to the elements into this constructor:

Array/ConcreteArrayProjection1d.h

```
template<class Subscriptor, class T>
ConcreteArrayProjection1d<Subscriptor, T>::
ConcreteArrayProjection1d(const ProjectionSubscriptor& s, T* t) :
    ConcreteArray1dRef<Subscriptor::ProjectionT, T>(s, t) {
}
```

This class, which is just an extension of the array reference class, passes its arguments on to its base class constructor, not shown (the analogous two-dimensional array reference class is shown on page 382).

At this point, we have a projection object, a temporary in our statement a[1]=1, as illustrated in Fig. 13.6. Although the projection object looks big in the figure, it only contains three data: two Subscript objects and a built-in pointer.

After the temporary is created in the statement a[1]=1, the temporary's assignment operator is called. This assignment operator passes the work on to the assignment operator in its base class:

Array/ConcreteArrayProjection1d.h

```
template<class Subscriptor, class T>
ConcreteArrayProjection1d<Subscriptor, T>&
ConcreteArrayProjection1d<Subscriptor, T>::operator=(const T& rhs) {
    ConcreteArray1dRef<ProjectionSubscriptor, T>::operator=(rhs);
    return *this;
}
```

The base class's assignment operator simply loops over the elements doing the appropriate element-by-element assignments:

Array/ConcreteArray1d.c

```
template<class Subscriptor, class T>
ConcreteArray1dRef<Subscriptor, T>&
ConcreteArray1dRef<Subscriptor, T>::operator=(const T& rhs) {
    Subscript n = shape(0);
    while (n-- > 0) (*this)(n) = rhs;
    return *this;
}
```

Despite appearances, these assignments set every third element of the array a, filling $a_{1,0}$, $a_{1,1}$, $a_{1,2}$, and $a_{1,3}$ with the value 1.0. This is a consequence of the strides

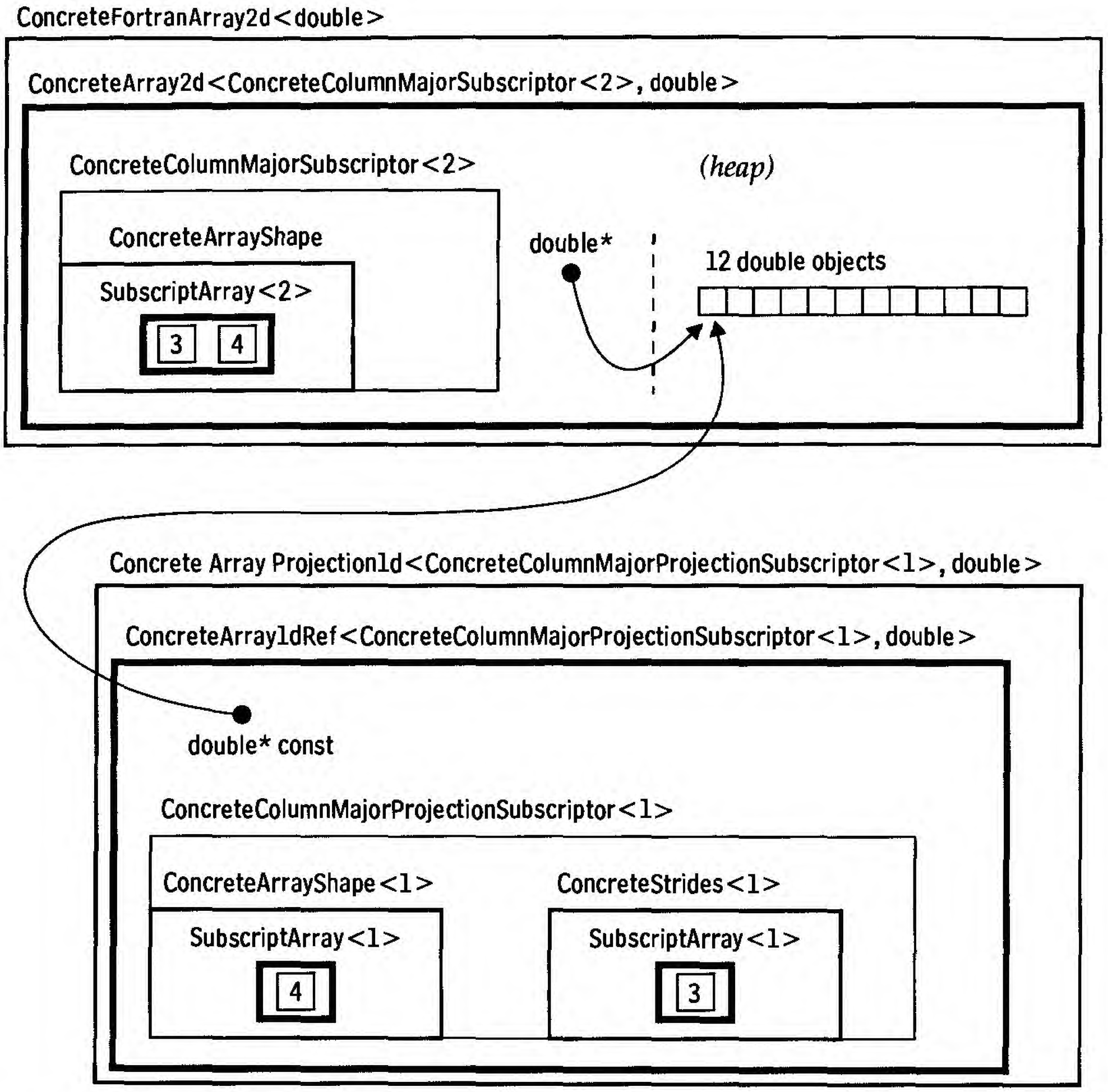


Figure 13.6 Sketch of the One-Dimensional Projection Object Created While Executing the Statement `a[1]=1` on Page 392.

built into the Subscriptor for the `ConcreteArrayProjection1d<Subscriptor, T>` temporary object created in `a[1]`.

13.5 Interfaced Arrays

The system of concrete array classes we have just described represents one approach to expressing commonality among array types. We combined common function names with the inline function and template expansion facilities of C++ to create array classes that share common implementation. To write client functions applicable to these arrays, we must select a fixed array type or write function

template clients. Such an approach yields implementation and name commonality while minimizing array-related runtime and space overheads.

Code size and compilation time are not minimized by this approach. Every kind of array results in different expansion of templates and compilation into different instructions. Expressing array commonality in source code means that the compiled code carries the burden of array variety.

The template approach also prevents handling different kinds of arrays homogeneously at runtime. We cannot, for example, create a class analogous to the IVTester class on page 249 which holds a member that in some situations is a column-major array and in other situations is a row-major array. And we can't create a list of arrays with some being packed or column major or so on.

When these disadvantages weigh heavily, we must turn to interfaces to express commonality. Interfaces go beyond name commonality to insist on common function calls. By creating an interface and routing all function calls through the interface, client functions become independent of array type. They see a single array type, the type of the interface. Class data members or lists of arrays can mix different arrays objects because they only work through the common interface.

Section 11.4 introduced, in simplified form, the array interfaces we use. In the next section we revisit and expand those interfaces. When we turn to implementing the interfaces, we choose not to start from scratch but rather to build on the concrete array system. We develop wrapper classes that implement the interface base class functions by forwarding them to corresponding functions in a concrete array. Recognizing the commonality of such an operation—all of the concrete arrays have the same function names to be forwarded—we create templates to automate the wrapping.

Section 13.6 tackles projections of interfaced arrays. As we found with concrete arrays, projections bring in issues of reference that add complexity. The solution for interfaced arrays is at once the same and different from the solution for concrete arrays. It will give us an opportunity to explore further the issues that arise in using interfaces.

13.5.1 Array Interfaces Revisited

We simplified the array interfaces developed in Section 11.4 by omitting projections. The interfaces we develop here must provide projections and should express, via virtual function interfaces, the commonality that exists among our concrete array classes. In other words, we want to express with virtual function interfaces what we expressed with name commonality in the concrete array classes, omitting all implementation details, including subscriptors.

Pure virtual functions in an interface base class must be defined in all implementation classes using functions with identical arguments and compatible return types. This requirement means that the more functions we add to the interface, the more likely it is that some kinds of implementations will be excluded. For example, if we add a pure virtual function corresponding to firstDatum() in ConcreteArray2d<Subscriptor, T>, then our interface cannot be attached to sparse arrays stored as lists or arrays accessed on remote machines. If we need an interface that specifies firstDatum(), a separate interface class derived from Array2d<T> should be added rather than restricting the generality of the Array2d<T> interface.

Adding virtual to functions that we do keep in the array interface also affects the return types of these functions. On page 375 we declared the project() functions to return objects of type ProjectionT. For the concrete class, each different subscripting algorithm was mapped into a compatible projection subscripting algorithm by template expansion. For example, the projections of ConcreteFortranArray2d<T> will be objects from ConcreteColumnMajorProjection1d<T>, whereas projections of ConcreteFormedArray2d<T> return row-major projection objects. An interface can specify only one return type for all category members. Since projection subscripting varies among array formats, we must conclude that the return type for projections must be an interface to projections.

As simple as this sounds, projections are objects and interfaces are not. We can't return an interface, only a reference (or pointer) to an interface. Since the projection function is called against an array that need not contain an internal projection object, we have no object to refer or point to. Instead we create and return an object with an array interface. For concrete arrays, this will be a wrapper object with the array interface forwarded to the concrete projection object. We call such a wrapper object an *accessor* to indicate that it accesses an internal object through a standard interface. These accessor objects are returned by value, but they function like references.

This description parallels the description of the concrete array references in Section 13.3. Like the concrete references, accessor objects must support references to const array objects. Since our projection accessor will have array interfaces, and since we will have both const and non-const projection accessors, we conclude that we need both const-like and non-const-like array interfaces.

The class DAG for two-dimensional array interfaces and accessors is shown in Fig. 13.7. Again we shall work from the top to the bottom. The ArrayShape interface base class appears on page 315.

Here is the const-like interface for two-dimensional arrays:

Array/Array2d.h

```
template<class T>
class ConstArray2d :
    public virtual ArrayShape {
```

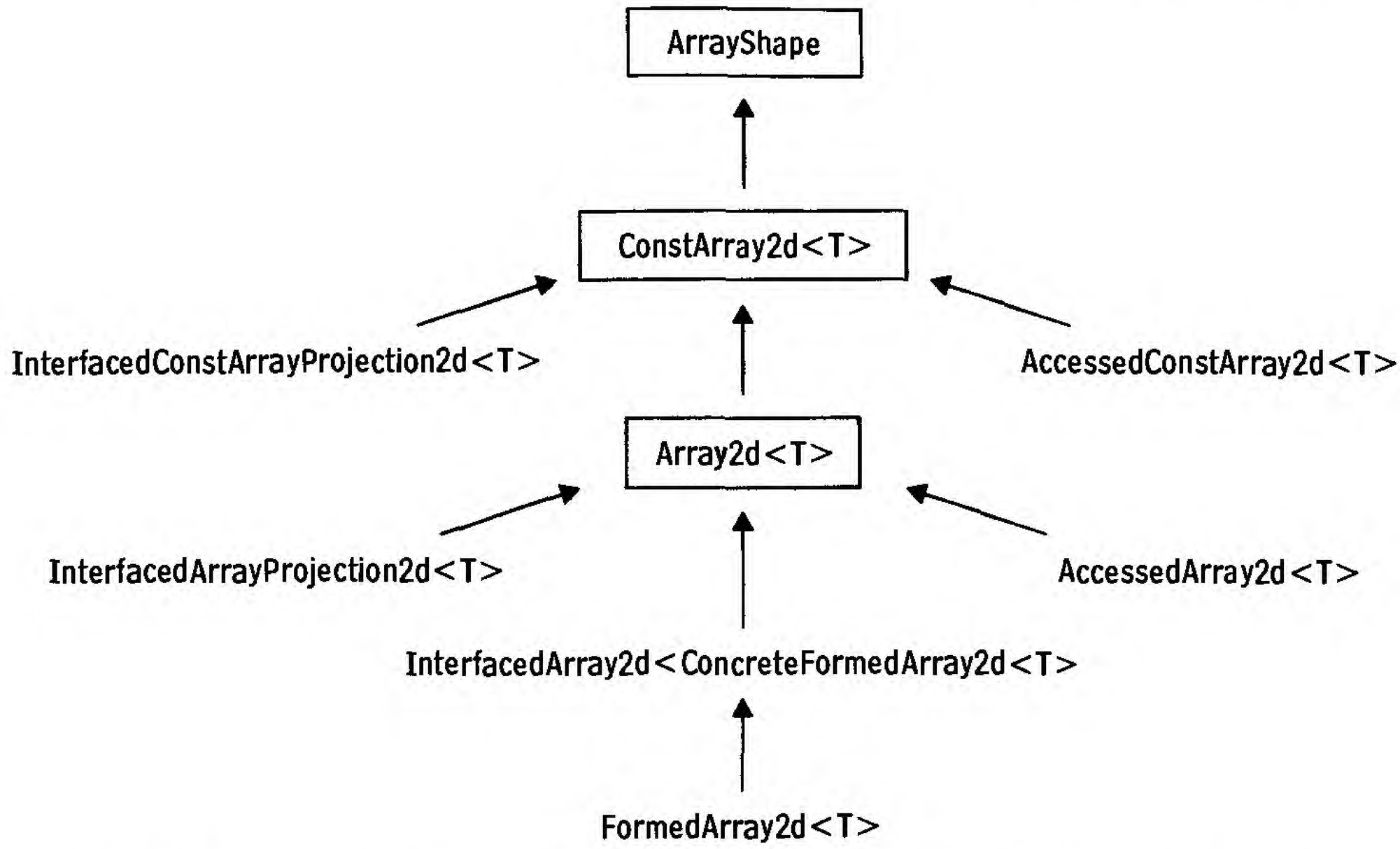


Figure 13.7 Class DAG for FormedArray2d<T>, Showing Two-Dimensional Array Interfaces, Projections, and Accessors.

```
public:
    typedef T EltT;

    typedef ArrayBrowser2d< ConstArray2d<T> > BrowserType;
    typedef AccessedConstArray1d<T>           ConstProjectionT;

    virtual const T&          operator()(Subscript i, Subscript j)     const = 0;
    virtual ConstProjectionT project(Subscript i, Dimension d = 0) const = 0;

    virtual Dimension dim() const { return 2; }

    virtual ConstProjectionT operator[ ](Subscript i) const;
    virtual ConstProjectionT row(Subscript i)         const;
    virtual ConstProjectionT column(Subscript i)      const;
private:
    void operator=(const ConstArray2d<T>&);    // Prohibit
};
```

Compare the ConstArray2d<T> interface with the ConcreteArray2dConstRef<Subscriptor, T> class (page 380). Strip out everything in the concrete array reference having to do with implementation, add virtual to the remaining member functions, and the result will be very similar to the ConstArray2d<T> interface. The definitions of the

various projection-related functions (not shown) all call project(), as in the concrete classes.

The private declaration of the assignment operator prevents C++ from generating an assignment operator that does member-by-member assignment. What members? That is the point: Without this declaration, the following code compiles:

ch13/bad-assign.C

```
void f(ConstArray2d<int>& a, ConstArray2d<int>& b) {
    a = b;
}
```

Since there are no members, the assignment has no effect and confuses the programmer, who thinks one array is being assigned to another. The assignment operator is not declared virtual because it is not to be overridden. Such an assignment operator can be hidden by a derived class that provides a useful assignment operator, but assignment through this interface will always be prohibited.

- Interface base classes should have either a virtual assignment operator, possibly pure virtual, or should declare a private non-virtual assignment operator.

The non-const two-dimensional array interface derives from the const interface, adding non-const subscripting and projections, and assignment:

Array/Array2d.h

```
template<class T>
class Array2d :
    public virtual ConstArray2d<T> {
public:
    typedef ArrayIterator2d< Array2d<T> > IteratorType;
    typedef AccessedArray1d<T>            ProjectionT;

    virtual const T&          operator()(Subscript i, Subscript j)     const = 0;
    virtual T&                operator()(Subscript i, Subscript j)           = 0;
    virtual ProjectionT       project(Subscript i, Dimension d = 0)          = 0;
    virtual ConstProjectionT project(Subscript i, Dimension d = 0) const = 0;

    virtual ConstProjectionT operator[ ](Subscript i) const;
    virtual       ProjectionT operator[ ](Subscript i);
    virtual ConstProjectionT row(Subscript i)         const;
    virtual       ProjectionT row(Subscript i);
    virtual ConstProjectionT column(Subscript i)      const;
    virtual       ProjectionT column(Subscript i);

    virtual Array2d<T>& operator=(const ConstArray2d<T>& rhs);
    virtual Array2d<T>& operator=(const T& rhs);
};
```

The assignment operators can be implemented straightforwardly using the interface:

Array/Array2d.c

```
template<class T>
Array2d<T>& Array2d<T>::operator=(const ConstArray2d<T>& rhs) {
    Subscript i = shape(0);
    if (i != rhs.shape(0)) throw ArrayErr::Shape();
    while (i-- > 0) (*this)[i] = rhs[i];
    return *this;
}

template<class T>
Array2d<T>& Array2d<T>::operator=(const T& rhs) {
    Subscript i = shape(0);
    while (i-- > 0) (*this)[i] = rhs;
    return *this;
}
```

Selecting the functions for our array interfaces does not provide us with any arrays. For implementation, we create wrapper classes around our concrete arrays.

13.5.2 InterfacedArray2d<AnArray2d>

Having gone to some effort to create consistent concrete and interfaced array systems, we can now reap the benefits. To implement the array interface with column-major subscripting, we create a class that derives from the interface and forwards each interface function to the corresponding function in a concrete array with column-major subscripting. Similarly, for row-major subscripting we forward to a concrete array with row-major subscripting, and so on for packed symmetric arrays and the rest.

We do not need to repeat this effort for each case. Name commonality in the concrete array system allows us to define a class template that wraps a concrete object in an interface and forwards calls from the interface to the concrete object:

Array/InterfacedArray2d.h

```
template<class ConcreteArray2d>
class InterfacedArray2d :
    public virtual Array2d<ConcreteArray2d::EltT> {
public:
    InterfacedArray2d(Subscript s0, Subscript s1) : the_concrete(s0, s1) {}
    InterfacedArray2d(const ConstArray2d<EltT>& a) :
                                        the_concrete(a.shape(0), a.shape(1)) { *this = a; }
    InterfacedArray2d(const ConcreteArray2d& a) : the_concrete(a) { }
```

```
    virtual Subscript shape(Dimension d) const { return the_concrete.shape(d);   }
    virtual Subscript numElts()          const { return the_concrete.numElts(); }

    virtual const EltT& operator()(Subscript i, Subscript j) const { return the_concrete(i, j); }
    virtual       EltT& operator()(Subscript i, Subscript j)       { return the_concrete(i, j); }

    virtual ConstProjectionT project(Subscript i, Dimension d = 0) const;
    virtual ProjectionT      project(Subscript i, Dimension d = 0);

    virtual Array2d<EltT>& operator=(const ConstArray2d<EltT>& rhs) {
        return Array2d<EltT>::operator=(rhs);
    }
    virtual Array2d<EltT>& operator=(const EltT& rhs) {
        the_concrete = rhs;
        return *this;
    }
protected:
    ConcreteArray2d the_concrete;
};
```

The template parameter, ConcreteArray2d, is purely symbolic and is only connected to the ConcreteArray2d<Subscriptor, T> through our use of the same function names in the class and in the template. Once again, the EltT typedef (in ConstArray2d<T>) allows us to use the element type as a parameter of the base class, Array2d<T>, and in the function return types without having to pass the element type as a separate template parameter.

For convenience, the functions that return simple types or pointers are given directly in the template body. Since assignment operators are not inherited, we must define them in this class: They forward the work as appropriate. The project() functions are defined in Section 13.6.2.

13.5.3 FormedArray2d<T>

Given the templatized interfacer of the preceding section, we can create an interfaced array from every concrete array. For example, given ConcreteFormedArray2d<T> defined like ConcreteFortranArray2d<T> but using a ConcreteRowMajorSubscriptor<2>, we get FormedArray2d<T>:

Array/FormedArray2d.h

```
template<class T>
class FormedArray2d :
    public InterfacedArray2d< ConcreteFormedArray2d<T> > {
```

```
public:
    FormedArray2d(Subscript n0, Subscript n1) :
                            InterfacedArray2d< ConcreteFormedArray2d<T> >(n0, n1) {}
    FormedArray2d(const ConstArray2d<T>& a) :
                            InterfacedArray2d< ConcreteFormedArray2d<T> >(a) {}

    virtual void reshape(const SubscriptArray<2>& s) { the_concrete.reshape(s); }

    virtual Array2d<T>& operator=(const ConstArray2d<T>& rhs) {
        return InterfacedArray2d< ConcreteFormedArray2d<T> >::operator=(rhs);
    }
    virtual Array2d<T>& operator=(const T& rhs) {
        return InterfacedArray2d< ConcreteFormedArray2d<T> >::operator=(rhs);
    }

    virtual const T* firstDatum() const { return the_concrete.firstDatum(); }
    virtual       T* firstDatum()       { return the_concrete.firstDatum(); }
};
```

We have extended the base class by including access to the array pointer in the underlying concrete array via the firstDatum() function.

13.6 Projections of Interfaced Arrays

Implementing interfaced arrays was simply a matter of forwarding function calls, and we glossed over the return type of the projections. In this section we go back and look at this aspect in detail.

13.6.1 AccessedArray2d<T>

As we described in Section 13.5.1, the projections of an interfaced array must be interfaced objects returned by value. Concrete projection objects cannot be returned: They have different types according to the format of the array data. Built-in references to an interface would have a single type, but they cannot be returned: They must refer to an object, and no object representing a projection lives inside of an array. Our solution is to create an interfaced referencelike object that controls the lifetime of a projection but hides its creation type behind an array interface.

We call the lifetime controlling object an accessor. An AccessedArray2d<T> holds a pointer to an Array2d<T> interface and controls the lifetime of any object in the Array2d<T> interface category:

Array/AccessedArray2d.h

```
template<class T>
class AccessedArray2d :
    public virtual Array2d<T> {
public:
    AccessedArray2d(Array2d<T>* newed_a) : the_array(newed_a) {}
    ~AccessedArray2d() { delete the_array; }

    virtual Subscript shape(Dimension d) const { return the_array->shape(d); }
    virtual Subscript numElts()          const { return the_array->numElts(); }

    virtual const EltT& operator()(Subscript i, Subscript j) const { return (*the_array)(i, j); }
    virtual       EltT& operator()(Subscript i, Subscript j)       { return (*the_array)(i, j); }

    virtual ConstProjectionT project(Subscript i, Dimension d) const {
        const Array2d<T>* const_array = the_array;    // Invoke const version of project
        return const_array->project(i,d);
    }
    virtual ProjectionT project(Subscript i, Dimension d) { return the_array->project(i, d); }

    virtual Array2d<T>& operator=(const ConstArray2d<T>& rhs) {
        return *the_array = rhs;
    }
    virtual Array2d<T>& operator=(const T& rhs) {
        return *the_array = rhs;
    }
private:
    Array2d<T>* the_array;

    AccessedArray2d(const AccessedArray2d<T>&);    // Prohibit copy
};
```

These accessor classes are themselves in the `Array2d<T>` interface category and therefore provide array behavior. Every member function calls the equivalent function through the contained pointer `the_array`. Whether those actions will be that of a row-major, column-major, or packed array, or a projection of one of these kinds of arrays, is determined by the array to which `the_array` points.

The accessor acts almost like a reference to an `Array2d<T>` interface. An `Array2d<T>&` would also act like an array and transparently handle all different kinds of array subscripting implementation. Unlike a reference, however, the ac-

cessor object automatically deletes the array or projection object when the accessor is destroyed. Likewise, an AccessedConstArray2d<T> object (not shown) controls the lifetime of any object in the ConstArray2d<T> interface category.

13.6.2 How Projections of Interfaced Arrays Work

The accessor gives us lifetime control, a uniform type to be returned by the virtual project() function in the Array2d<T> interface, and an array interface through which to access the elements. The final step in implementing projections of interfaced arrays is to create an accessor that points to a projection object whose creation type varies to match the format of the array being projected, but that has an array interface. For this purpose, we introduce an interfaced projection class:

Array/InterfacedArray2d.h

```
template<class ConcreteArray2d>
class InterfacedArrayProjection1d :
    public virtual Array1d<ConcreteArray2d::EltT> {
public:
    InterfacedArrayProjection1d(const ConcreteArray2d::ProjectionT& a) :
        the_concrete(a.subscriptor(), a.firstDatum()) {
    }

    virtual Subscript shape(Dimension d) const { return the_concrete.shape(d);   }
    virtual Subscript numElts()          const { return the_concrete.numElts(); }

    virtual const EltT& operator()(Subscript i) const { return the_concrete(i); }
    virtual       EltT& operator()(Subscript i)       { return the_concrete(i); }

    virtual Array1d<EltT>& operator=(const ConstArray1d<EltT>& rhs) {
        return Array1d<ConcreteArray2d::EltT>::operator=(rhs);
    }
    virtual Array1d<EltT>& operator=(const EltT& rhs) {
        the_concrete = rhs;
        return *this;
    }
protected:
    ConcreteArray2d::ProjectionT the_concrete;
};
```

This class wraps an interface around the projection object returned by the specified concrete array class. The connection between the concrete array class and

the corresponding projection object is all expressed symbolically, via the ProjectionT typedef provided by all concrete arrays.

The interfaced concrete projection object can be controlled by an accessor object and returned by project():

Array/InterfacedArray2d.h

```
template<class ConcreteArray2d>
InterfacedArray2d<ConcreteArray2d>::ProjectionT
InterfacedArray2d<ConcreteArray2d>::project(Subscript i, Dimension d) {
    Array1d<EltT>* p =
        new InterfacedArrayProjection1d<ConcreteArray2d>(the_concrete.project(i, d));
    return AccessedArray1d<EltT>(p);
}
```

The interfacer template gets the projection object by calling project() on the concrete array it holds by the member datum the_concrete, passes it to InterfacedArrayProjection1d<ConcreteArray2d> to be wrapped up in an interface, and puts a pointer to the interface into the accessor, which is then returned.

To recapitulate, let's look at the execution of the following code:

ch13/tCollect.C

```
FormedArray2d<double> a(3, 4);
a[1] = 1;
```

The projection operator called by a[1] causes creation of a temporary ConcreteArrayProjection1d<Subscriptor, T> object, with Subscriptor set to ConcreteRowMajorArraySubscriptor<2> and T set to double. This concrete projection object points to the data for the first row of the array a. The projection object is then wrapped in an Array1d<T> interface by InterfacedArrayProjection1d<ConcreteFormedArray2d<double> >, and the lifetime of this object is controlled by the AccessedArray1d<T> that holds a pointer to that interface. The assignment comes into the AccessedArray1d<T> object and gets forwarded to the internal pointer, which triggers a second function call forwarding to the concrete array projection.

13.7 Iterators

Given a collection of elements, it is often necessary to be able to examine or modify every element. The process of accessing the elements is called *iteration*. Traditionally, iteration is encoded in the control structures of a program. An *iterator* is an object that controls iteration. We introduce our style of iteration objects in this section, giving examples of their use and definitions of iterators for our con-

crete and interfaced array systems. These definitions illustrate parallel use of these two systems that we hope will help clarify their relationship.

13.7.1 Iteration without Iterators

To understand the need for iterator objects, we start by looking at iteration as used in conventional FORTRAN and C or C++ programs. FORTRAN programs iterate over array elements using loops. For example, to find the Frobenius norm of a matrix (square root of the sum of the squares of the elements), one might write

ch13/do.f

```
      double precision function fronrm(array, m, n)
      integer m, n
      double precision array(m, n), scale, sumsq, absx

C     The sum of the squares is scale**2 * sumsq:
      sumsq = 0.0
      scale = 1.0

      do 100 j = 1, n
            do 200 i = 1, m
                  absx = abs(array(i, j))
                  if (absx .gt. scale) then
                        sumsq = (scale / absx)**2 * sumsq + 1
                        scale = absx
                  else
                        sumsq = sumsq + (absx / scale)**2
                  end if
200         continue
100   continue
      fronrm = scale * sqrt(sumsq)
      end
```

This iteration uses a random access operator—subscripting—to access an array in a sequential order. Logically, the order of access is irrelevant and the code overspecifies the computation; efficiency considerations suggest that the elements be accessed in the order they are stored in memory, as we have written it. (The same loop written in a C or C++ program would need to reverse the roles of the loop variables i and j to access the array in memory order.)

The equivalent program can be written directly in C++ without using the random access subscript operation by using pointer arithmetic to access the elements in sequential order:

ch13/while.C

```
double FrobeniusNorm(double* array, int n) {
    double sumsq = 0.0;
    double scale = 1.0;
    double* end = array + n;
    for (double* cur = array; cur < end; cur++) {
        double absx = fabs(*cur);
        if (absx > scale) {
            sumsq = sqr(scale / absx) * sumsq + 1;
            scale = absx;
        }
        else sumsq += sqr(absx / scale);
    }
    return scale * sqrt(sumsq);
}
```

This code assumes that the array elements are stored contiguously in memory and would not work for arrays stored otherwise; see Exercise 13.11. In fact, many FORTRAN compilers will optimize the two nested loops and subscript calculation in the preceding example into code that steps a pointer, as we have written this example.

The FORTRAN example overspecifies the computation both by fixing the array type and by using random access for a sequential operation. The C example has neither problem but expresses the computation in terms of low-level details.

13.7.2 Using Array Iterators

Iterators put the details of iterating over the elements of a collection into an object. Iterators can be written to provide access to array elements in any desired order and for any array representation. They don't overspecify the computation and they don't require assumptions about efficiency to be distributed throughout the code. Our iterators provide a more() member function to test for the end of the sequence of elements, advance() to advance to the next element, and current() to access the current element. Here is a version of the Frobenius norm function written using an iterator:

ch13/IteratorFrobenius.C

```
template<class Array>
Array::EltT FrobeniusNorm(const Array& a) {
    Array::EltT sumsq = 0;
    Array::EltT scale = 1;
    for (Array::BrowserType i(a); i.more(); i.advance()) {
        Array::EltT absx = abs(i.current());
```

```
            if (absx > scale) {
                sumsq = sqr(scale / absx) * sumsq + 1;
                scale = absx;
            }
            else sumsq += sqr(absx / scale);
        }
        return scale * sqrt(sumsq);
}
```

This version handles arrays of arbitrary dimensionality and arbitrary storage layout. The array must provide a nested class or typedef for the name BrowserType, specifying the type of the object that knows how to iterate the elements of the array. We call it a *browser* rather than an iterator to connote read-only access to the elements. (The name IteratorType specifies the corresponding iterator object.)

The browser object is constructed from the array whose elements are to be examined, and then the browser is used to control the loop. There is only a single loop, regardless of the array's dimensionality. All knowledge of the array's storage layout and how to access the elements is encapsulated in the browser object. Moreover, there is no opportunity to make the careless errors that are common when writing nested loops (e.g., testing the wrong loop variable): Once the browser implementation is debugged, the same implementation is reused in all code that needs to examine all array elements.

Iterators hide the representation of their associated collections, and they let us code only what we need: They respond to "are there any more elements?" and "next, please."

- Use iterators and browsers, not subscripting, to express sequential access to all the elements of an array.

Notice that client code for iterators is templatized to allow a spectrum of array types to be used by name commonality. This allows inline functions to apply to the iterators for concrete arrays to reduce runtime overheads. For interfaced arrays, we still obtain the code size advantage when we use iterators: All of the interfaced arrays will use a single iterator type and all client template functions will be expanded for this single iterator type.

13.7.3 Implementing Concrete Array Iterators

All of our concrete arrays store their elements as numElts() contiguous objects (i.e., as a built-in array of objects). From a pointer to the first such object and the numElts() function, we can construct an iterator that mimics the pointer manipulations in the version of FrobeniusNorm() on page 408.

For const objects, the read-only or browser implementation looks like this:

Array/ConcreteArrayIterator.h

```
template<class ConcreteArray>
class ConcreteArrayBrowser {
public:
    ConcreteArrayBrowser(const ConcreteArray& anArray) :
        cur(anArray.firstDatum()),
        end(cur + anArray.numElts()) {
    }
    Boolean                          more()    const { return cur < end; }
    void                             advance()       { cur++; }
    const ConcreteArray::EltT& current() const { return *cur; }
private:
    const ConcreteArray::EltT* cur;
    const ConcreteArray::EltT* end;

    friend class ConcreteArrayIterator<ConcreteArray>;
    ConcreteArrayBrowser(const ConcreteArray::EltT* c, const ConcreteArray::EltT* e) :
        cur(c),
        end(e) {
    }
};
```

A pointer to the current element and a pointer to the last element are contained internally; they are initialized from firstDatum() and numElts() in the constructor. The cursor pointer cur is analogous to the cur pointer in the version of FrobeniusNorm() on page 408. Since we have to store either a count or a pointer in our iterator object, we store a pointer to the first location beyond the last element to avoid
M §5.7 arithmetic on a count. (If not dereferenced, a pointer can point to the first location beyond the last element of a built-in array, but no further.) The functions are inlined here on purpose. Because they are quite simple, we can reasonably expect inline expansion in most compilers.

Since a browser provides read-only access to an array's elements and an iterator provides read-write access, it should be possible to convert an iterator to the corresponding browser. That way, if you have an iterator over an array, you can pass it to a function that expects a browser over the array.

- Write browser-iterator pairs with conversion from the iterator to the browser.

The private constructor in ConcreteArrayBrowser<ConcreteArray> is intended to be used by a conversion operator in the iterator class; the friend declaration provides access to the iterator class:

Array/ConcreteArrayIterator.h

```
template<class ConcreteArray>
class ConcreteArrayIterator {
public:
    ConcreteArrayIterator(ConcreteArray& anArray) :
        cur(anArray.firstDatum()),
        end(cur + anArray.numElts()) {
    }
    Boolean more()                    const { return cur < end; }
    void advance()                          { cur++; }
    ConcreteArray::EltT& current() const { return *cur; }

    operator ConcreteArrayBrowser<ConcreteArray>() {
        return ConcreteArrayBrowser<ConcreteArray>(cur, end);
    }
private:
    ConcreteArray::EltT* cur;
    ConcreteArray::EltT* end;
};
```

13.7.4 Implementing Interfaced Array Iterators

For interfaced arrays, we cannot access a firstDatum() pointer because the array may not have any objects in memory to point to. Certainly, advance() cannot use pointer addition. The ConcreteArrayIterator<T> is also useless for projections since we have to skip around in the contiguous storage by amounts that depend on the kind of projection and the dimension projected away (consider projections of packed upper triangular matrices). Using subscripting, which is available to all arrays, we can implement array iterators without knowing how arrays are implemented. This is the approach we use here.

Let's consider the two-dimensional case, looking at the nested loops we would write if not using iterators:

ch13/tCollect.C

```
Subscript n0;    // Upper bound on s0
Subscript n1;    // Upper bound on s1
// ... set n0 and n1
for (Subscript s0 = 0; s0 < n0; s0++) {
    for (Subscript s1 = 0; s1 < n1; s1++) {
        // ... do something ...
    }
}
```

These loops operate only on subscripts and are independent of array element type. Similarly, the subscript manipulations for our iterators can be held in an element-type independent base class:

Array/ArrayIterator2d.h

```
class ArrayStepper2d {
public:
    ArrayStepper2d(Subscript shape0, Subscript shape1) :
        s0(0), s1(0),
        n0(shape0), n1(shape1) {
    }
    Boolean more() const { return s0 < n0; }
    void advance() {
        if (++s1 >= n1) { s1 = 0; s0++; }
    }
protected:
    Subscript s0;
    Subscript s1;
    const Subscript n0;
    const Subscript n1;
};
```

Given this class, the equivalent loop skeleton can be written like this:

ch13/tCollect.C

```
for (ArrayStepper2d i(n0, n1); i.more(); i.advance()) {
    // ... do something ...
}
```

The only question is what to do in the loop. The answer is to add a current() member by derivation and to use it to perform the required action:

Array/ArrayIterator2d.h

```
template<class Array>
class ArrayBrowser2d :
    public ArrayStepper2d {
public:
    ArrayBrowser2d(const Array& anArray) :
        a(anArray),
        ArrayStepper2d(anArray.shape(0), anArray.shape(1)) {
    }
    const Array::EltT& current() const { return a(s0, s1); }

    ArrayBrowser2d(const Array& anArray, const ArrayStepper2d& step) :
        a(anArray),
        ArrayStepper2d(step) {
    }
private:
    const Array& a;
};
```

This class template is parameterized by the kind of array being iterated over; setting the parameter to a specific kind of array provides the definition of iteration for that kind of array; setting the parameter to an array interface, like `Array2d<T>`, provides the definition of iteration for all members of the interface category. Indeed this is how the array interfaces define their browsers and iterators; see, for example, the definition of `ConstArray2d<T>` on page 398.

The browser holds a reference to the array, and the browser's `ArrayStepper2d` subobject holds the state of the iteration. Both are initialized by the constructor. All that remains for the current() member to do is to return the element selected by the current state (subscripts).

The corresponding iterator object is almost identical, requiring only that a few `const` keywords be omitted and a conversion to browser operator be added. We do not show the code here (see Exercise 13.17).

Although using subscripting for iteration uses a random access operation for a sequential process, our client functions do not know this. In particular, we can write client functions that use iterators uniformly for interfaced arrays and for all kinds of lists and other kinds of collections.

Iterators are important for both interfaced arrays and concrete arrays, but for different reasons. For concrete arrays they help us produce efficient abstracted code; for interfaced arrays they help us abstract beyond arrays to allow us to write clients applicable to a wide range of collections.

13.8 Summary

We have described in detail a system of concrete arrays supplying name commonality and we implemented these arrays by exploiting structural commonality. The system supports any kind of array that can be stored as contiguous objects, allowing different formats for the elements. We have also described a system of interfaced arrays that hides all of the differences among arrays, including element storage layout, whether or not elements are stored contiguously, or even if elements are stored in memory. Finally we described iterators that allow traversal of arrays as collections without consideration of the array shape.

Obviously our system of concrete array classes, projection classes, references, subscriptors, and so on is overkill if your goal is to write C++ programs for FORTRAN-like arrays of `double` numbers. However, exactly the same code works for any element type `T`, and much of the code and all of the structure works for row-major or packed storage arrays. Moreover, client functions or classes written for this array system will work for a wide variety of arrays and object types. Selecting new trade-offs and using existing code written for specific array formats will be easier within a comprehensive framework like the one we have built here.

The multiple levels of objects and function calls in this system are an inevitable consequence of exploiting commonality. If you want to avoid recoding the same function with slight variations, you must call a common function with variable arguments; if you want to avoid reconstructing the same object with slight variations, you must layer variations on common base classes. In our concrete system, all of the variations are expressed via templates: At compile time a good compiler can collapse most of the structure, yielding good performance.

As an example of a system of concrete classes, our concrete arrays illustrate issues that arise when this route is used to support programming. Use concrete classes as low-level building blocks, or where performance is absolutely critical. Here is a rule of thumb:

- An object that you wish were built into the language is a good candidate for a concrete class.

Introducing a concrete class is also appropriate when measurements show a specific class to be a performance bottleneck.

As with the concrete array system, the interfaced system may seem rather elaborate, especially the facility for array projection including the expression of constness. Our aim is to illustrate the issues, and in this regard the example is typical of programming through virtual function interfaces. The interface base class must operate on nonspecific types, and when objects of specific types are created to implement the virtual functions of an interface, these objects must be returned using only pointers or references to their interfaces, while at the same time their lifetime control must pass out of the virtual function. Consequently some memory management—accessors in this case—is required.

The overhead of the interfaced arrays may or may not be an important issue in real code. Our explanations here went into detail, but suppose for the moment that you did not read this chapter. Instead we return to the level of description in Chapter 11, where we gave the functions allowed for `Array2d<T>` objects and the construction arguments for various arrays like `FortranArray2d<T>` and `FormedArray2d<T>`. Then we can write client code to perform many array operations simply and without regard to details. Those parts of the code that proved too slow could be improved to close to the level of any manually created C++ code by selectively replacing the interfaced arrays and interface function calls with corresponding concrete arrays and template functions. The advantage of a systematic class system lies in the ability to pursue one or another trade-off easily rather than relying on the success of a single design decision.

13.9 Notes and Comments

13.1 Both C++ and FORTRAN arrays are inadequate for the full range of scientific programming activities. FORTRAN-90 revamps arrays and adds dynamic memory allocation

[78, 79]. The C++ approach provides the facilities to define appropriate array types but leaves out the definitions. Our array classes are neither complete nor industrial strength; they are a vehicle for illustrating important C++ concepts and techniques. Although they are more complete than the built-in C++ arrays, they do not have the breadth of storage schemes valuable to broad application, nor have they been tuned for performance.

13.2 Providing the Const versions of the concrete array reference classes and the array interface classes increases the number of classes. We think that the benefit outweighs the added complexity because working with const objects is part of C++, so array classes should support that style. Moreover, since we have not seen the relevant techniques elsewhere, we wanted to illustrate how const objects can be accommodated. The scheme propagates simply once you understand it.

13.3 Smalltalk-80 [49] provides classes for various collections in addition to arrays. The NIH class library [51] provides similar classes for C++, in a Smalltalk style.

13.4 Iterators are an example of a *programmer-defined control structure*, the control structure analog of a programmer-defined type. Programmer-defined control structures appear prominently in languages, like CLU, that focus on abstract data types. See [48, Section 5.1] or [92, Section 6–10], and papers referenced therein, for more detail.

13.5 The name *accessors* comes from [64]. They are called *envelope classes* in [27] and *delegation* classes by others.

13.10 Exercises

13.1 Most of the class templates described in Section 13.2 are parameterized by dimensionality. Use template specialization to supply dimension-specific classes tuned for performance. Determine experimentally whether your tuning makes a significant performance difference.

13.2 Consider the subscriptor() member of ConcreteArray2d<Subscriptor, T> defined on page 375. Given that Subscriptor is a private base class of ConcreteArray2d<Subscriptor, T>, why is the expression *this a legal return value? Explain how the returned value is created.

13.3 Define a ConcreteElasticArray2d<T> class template that provides two-dimensional arrays that behave like concrete formed arrays except that their size is adjusted automatically by array assignment.

13.4 Define a ConcreteRigidArray2d<T, n0, n1> class template that provides two-dimensional arrays of size n0 by n1, with row-major storage layout. Discuss how this class template fits into the concrete array system developed in this chapter.

13.5 Consider maxArray2dElement() on page 365. There is no inherent reason that this code should be restricted to two-dimensional arrays: Finding the maximum element only

requires accessing each element, regardless of the array's dimensionality. Using iterators and browsers, rewrite the function so that it works with arrays of arbitrary dimensionality.

13.6 Can you reimplement the concreteCopy() function template on page 383 using browsers and iterators instead of projection? If so, do it. If not, explain why not and suggest an alternate approach.

13.7 Design and implement classes for one-dimensional concrete array reference objects.

13.8 The array classes of Section 13.2 provide 0-origin subscripting consistent with C++'s built-in arrays. How could you alter the classes to provide 1-origin subscripting consistent with FORTRAN's arrays? Discuss the efficiency implications, if any, of your changes.

13.9 Revisit Exercise 13.8, this time providing classes for both FORTRAN-like and C++-like arrays. Can you do this while neither altering nor duplicating the code of Section 13.2? Discuss the execution efficiency of your solution. Also discuss the programmer efficiency of your solution.

13.10 Write a class DiagonalOfArray2d<T> that can be used like this:

ch13/tDiagonal.C

```
template<class T>
void set_to_identity(Array2d<T>& a) {
    a = T(0);
    DiagonalOfArray2d<T> d(a);
    d = T(1);
}
```

Your class should be in one of the array interface categories. Does your class work with const arrays? If not, devise a solution.

13.11 The book *Numerical Recipies in C: The Art of Scientific Computing* [93, Section 1.2] advocates representing two-dimensional arrays in C by a pointer to a one-dimensional array of pointers to rows, each a one-dimensional array of elements. Implement such two dimensional arrays in C++. In what way can such a representation fit into our system of concrete arrays? Interfaced arrays?

13.12 Write input and output operators for concrete arrays.

13.13 Write input and output operators for interfaced arrays. Contrast your solution to this exercise with your solution to Exercise 13.12.

13.14 Although the interfaced arrays like FormedArray2d<T> use concrete arrays in their implementation, the interfaced and concrete array systems are not coupled. For example, you can't assign an interfaced array to a concrete array. Add this capability by writing one or more function templates, called arrayCopy(), that take two array arguments and copy the second argument into the first argument. Discuss the advantages and disadvantages of this means of coupling the two array systems.

13.15 Given numbers $x_0, \ldots, x_n$, the matrix

$$V(x_0, \ldots, x_n) = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & \cdots & x_n \\ \vdots & \vdots & & \vdots \\ x_0^n & x_1^n & \cdots & x_n^n \end{bmatrix}$$

is called a *Vandermonde matrix*. Such matrices arise in approximation and interpolation problems [50, Section 4.6]. Write a Vandermonde class that behaves like a Vandermonde matrix but only stores $n + 1$ numbers. Your class should fit into either the concrete array system, the interfaced array system, or both. Explain your design choice.

13.16 The array-to-array assignment operator shown on page 401 requires a virtual function call to access each element of both arrays (assuming that one-dimensional assignment of interfaced arrays is implemented similarly). For assignment of large arrays with small elements (e.g., built-in objects), this is expensive. Using the double dispatch scheme of Section 12.4.2, modify InterfacedArray2d<A> to provide more efficient array-to-array assignment in the case that both arrays are actually instances of InterfacedArray2d<A> for the same concrete array type A. Discuss the advantages and disadvantages of this approach.

13.17 Implement ArrayIterator2d<T>, a version of ArrayBrowser2d<T> that grants write access to iterated elements of interfaced arrays.

CHAPTER 14

Pointer Classes

POINTERS allow us to build sophisticated data structures, to work with dynamically allocated memory, and to work through interfaces. These uses make pointers vital to C++ programs, but the erroneous use of pointers is a significant source of program bugs. In fact, there are several successful commercial products aimed primarily at detecting bad pointer uses at runtime. In this chapter we describe classes that can make programming with pointers easier and less error prone.

The classes that we develop in this chapter are based on the observations that one C++ construct—the built-in pointer—is used for multiple purposes and that the built-in pointer type is both too powerful and too weak for each purpose. For example, a pointer to a single dynamically allocated object can be incremented or decremented even though this makes no sense—the pointer type is too powerful—but the pointer provides no assistance in controlling the lifetime of the object—the pointer type is too weak.

We begin in Section 14.1 with an overview of the common applications of pointers in C++ code, focusing on the use of pointers in aggregation in Section 14.2. The remainder of the chapter illustrates the design of programmer-defined pointer types with behaviors tailored to specific applications. Section 14.3 introduces the idea of pointer classes. Section 14.4 implements pointers that provide lifetime control of dynamically allocated objects. Section 14.5 implements pointers that manage sharing of an object among several other objects, and Section 14.6 gives an example that uses them. Section 14.7 implements pointers that provide lifetime control of dynamically allocated objects knowing only their interface, including the ability to copy objects.

14.1 Pointer Uses

Some of the ways that C++ pointers are used include the following:

Lifetime Control. The C++ `new()` operator returns a pointer to an object. This pointer or a copy of this pointer must be supplied to `delete` when we are done

with the object to recover the memory allocated. Once deleted, the object must not be deleted again.

Arrays. Manipulation of C++ built-in arrays exploits the equivalence, built into the language, between an array and a pointer to its first element. This equivalence is used both to allow arrays to be created dynamically and to step through an array's elements. Pointers can be incremented and—if the pointer points to an object in a built-in array—the incremented pointer will point to the succeeding object in the array. In other cases the result is undefined.

Referential Aggregation. The process of combining objects within another object is called *aggregation*, and the resulting object is called an *aggregate*. A class aggregates its member data objects. If a member datum is declared as an object, the aggregate *contains* the member object and we say we are using *containing aggregation*. If a member datum is declared as a pointer (or reference), only the pointer itself is contained; the object pointed to is *referenced* and we call this *referential aggregation*.

Cursors. Objects in a data structure can be remembered by maintaining a pointer to the object. The pointer can be reset to refer to another object in the data structure as a means of visiting objects in the structure.

Interfaces. Client functions and classes written knowing only the type of an interface base class can manipulate objects through pointers to their interface base (Chapter 9 and Section 12.2).

These uses of pointers are not mutually excluding. For example, the GPIB controller simulator of Section 9.9 (page 254) contains an array of `GPIBInstrumentSimulation` pointers. These pointers both point to interfaces and are used to control the lifetime of the contained objects.

Some uses of pointers are especially prone to error. The pointer returned by `new` when an array is allocated dynamically must be deleted with the `delete [ ]` operator, not the `delete` operator. Since the language doesn't distinguish among pointer uses, compilers can't diagnose use of the wrong deletion operator. In-

ARM §5.3.4

stead you get undefined runtime behavior, in some implementations failure to run destructors on elements of an array when `delete` is used on a pointer to an array and disaster when `delete [ ]` is used on a pointer to an object. We avoid this problem by using array classes, like those of Chapter 13, instead of built-in arrays.

Arithmetic on a pointer is always incorrect unless the pointer points to an element of an array. Again the language doesn't distinguish among the various uses of pointers and therefore compilers can't provide much help. We avoid this problem by preferring iterators over pointer arithmetic.

A large part of the remaining pointer problems come from improper implementation of referential aggregation. Therefore we now investigate referential aggregation in some detail.

14.2 Referential Aggregation

With referential aggregation, we have an object with a pointer member datum and the pointer points to another object, the *aggregated object*. See Fig. 14.1. When using referential aggregation, we must always face the issue of memory management: When the aggregate is deleted, will the aggregated object be deleted or not? If the aggregate controls the lifetime of the aggregated object, we say we are using *controlling aggregation*; if the aggregate does not control the lifetime of the aggregated object, we say we are using *shared aggregation*. The two kinds of aggregation—containing and referential—crossed with two kinds of lifetime control—controlling and sharing—give us four combinations. To understand the problems introduced by referential aggregation, it's useful to understand all four cases. Fig. 14.2 sketches these cases.

Containing-Controlling Aggregation. Containing aggregation of full objects is always controlling aggregation in C++: When the aggregate object is destroyed, all of its members are also destroyed automatically. Using member objects—controlling-containing aggregation—is the most straightforward form of aggregation and does not involve pointers.

Referential-Controlling Aggregation. As we saw in Section 7.6, a class can take control of the lifetime of a referenced object. The class `NoDangle` (Page 192) shows how to implement referential-controlling aggregation manually. The pointer member `int* p` refers to and controls the lifetime of an `int` object. Notice, however, that the code for the lifetime control is spread throughout the

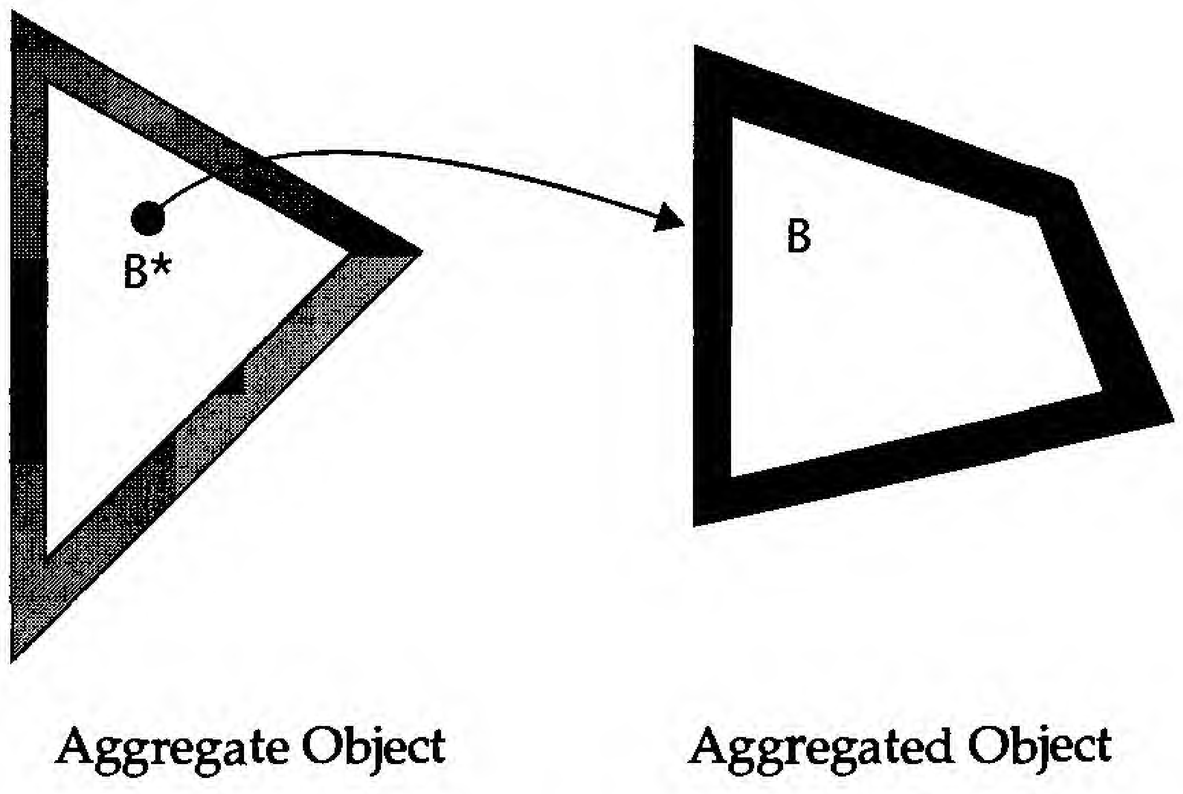


Figure 14.1 Illustration of Referential Aggregation.

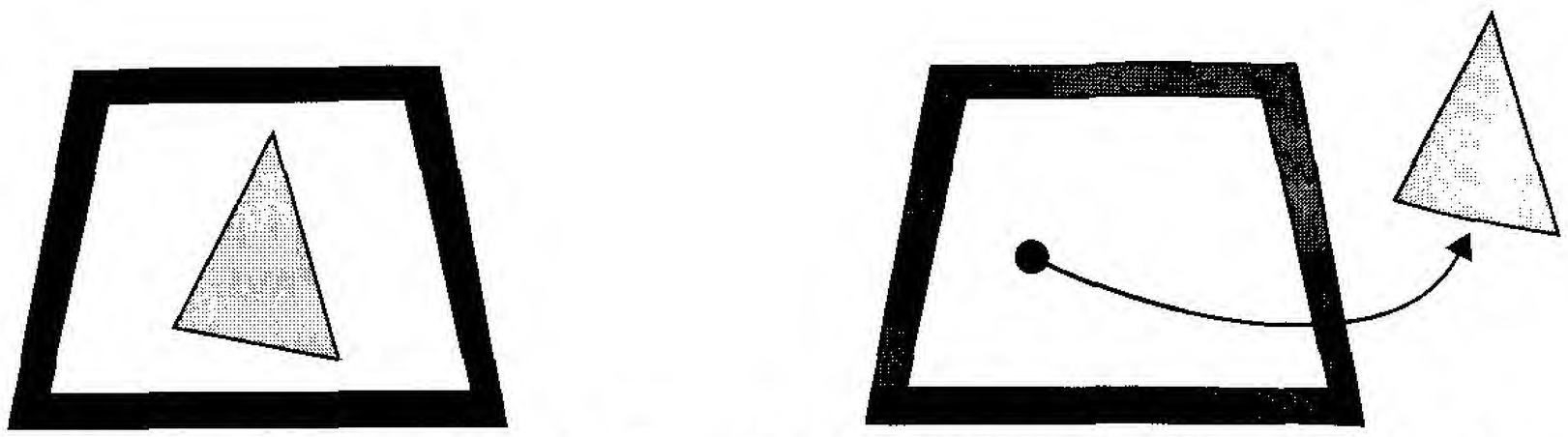


Figure 14.2 Four Cases of Aggregation. A containing-controlling aggregate has a component member datum object. A referential-controlling aggregate has a component member pointer that manages the lifetime of another object. A containing-sharing aggregate contains a member that it shares. A referential-sharing aggregate contains a pointer that points to an object shared with other aggregates.

class. In a more complex class it could be mixed with code for other purposes. With a built-in pointer member datum, we can't tell whether the pointer implements controlling or sharing aggregation unless we study the class's member function definitions. Using member pointers managed as in `NoDangle`—referential-controlling aggregation—is more involved than containing-controlling aggregation with member objects, but at least it is confined to a single class.

Containing-Sharing Aggregation. The iterators and projection objects of Chapter 13 illustrate this third combination. In both cases the array components are con-

tained in an array object, but through firstDatum() the contents are shared with another object. A pointer is used in the projection or iterator to refer to the shared data. Using member pointers to point to member objects in another aggregate—sharing-containing aggregation for the other aggregate—relies on the referenced aggregate outliving the reference. Unlike the controlling aggregation in NoDangle, code from at least two classes is involved (the array and iterator classes, for example). However, these two classes at least clearly identify the class with the controlling aggregate. For example, ConcreteArray2d<T> certainly controls lifetime. With experience we learn the coding idioms that ensure that the array is around when we use the iterator or projection.

Referential-Sharing Aggregation. You get referential-sharing aggregation when you use a built-in pointer as a member datum. The default copy behavior of built-in pointers—copy values—and the default copy behavior of classes—memberwise recursive copies—combine to make the default behavior of referential aggregation be sharing aggregation: When the aggregate is copied, both the original aggregate and the copy have a pointer to the same shared aggregated object. With sharing aggregation, we must coordinate lifetime control among multiple objects. A more general version of this problem occurs when an object is shared among instances of more than one class. Using built-in pointers for referential-sharing aggregation is difficult because we must determine by some ad hoc means which aggregate will control the lifetime of the component.

If we rely on built-in pointers to implement any of the last three aggregations (the ones that use pointers), we must remember to supply the appropriate behavior when aggregate objects are copied, assigned, and destroyed. Forgetting any one of them leads to failure. Even when we remember to supply an appropriate copy constructor, assignment operator, and destructor, we can still have a memory leak if the constructor for any of the aggregate's members throws an exception, as illustrated in the following code:

ch14/exceptionLeak.C

```
class Leaker {
public:
    Leaker(Subscript size) : p(new double(0)), array(size) {}
    Leaker(const Leaker& lkr) : p(new double(*lkr.p)) {}
    Leaker& operator=(const Leaker& lkr) {
        p = new double(*lkr.p);
        array = lkr.array;
        return *this;
    }
    ~Leaker() { delete p; }
```

```
private:
    double* p;
    ConcreteFormedArray1d<int> array;
};

void f() {
    // ...
    Leaker l(-3);
}
```

When f() is called, the constructor for Leaker is invoked with an argument of −3. The constructor allocates a double, saves the pointer to it in the member datum p, and then invokes the array constructor with argument −3. This fails because of the negative size and an exception is thrown. The exception causes constructed objects bound to the local variables of f() to be destroyed. However, since destructors for partially constructed objects are not run, the destructor for the Leaker instance is not run, leaving the double allocated by the constructor as garbage.

ARM §15.3c

Our approach to these problems is to design new pointer types for aggregation with pointers. After a general overview of pointer classes, we describe pointers for controlling-referential aggregation of creation types in Section 14.4. These pointers make such aggregation almost as easy and safe as member objects. Pointers for sharing-referential aggregation with lifetime control by reference counting are described in Section 14.5. These pointers are also easy to use and safe, but the reference counting adds overhead. Pointers for controlling-referential aggregation of objects known only through interface base types are described in Section 14.7. These pointers require a new virtual function in the base type, and this function introduces possible errors. Nevertheless these pointers are safer than other alternatives and allow powerful abstractions to be built.

14.3 Programmer-Defined Pointer Classes

Pointer objects are a kind of *smart pointer*: They intercept indirection and dereferencing, perform various actions, and then continue with the indirection or dereferencing. The pointer classes in this chapter exist to establish a relationship between the lifetime of the pointers and the lifetime of dynamically allocated objects. Establishing the relationship requires both code for the pointer classes and disciplined use of the classes. The code and the discipline concentrate and formalize the work of memory management for referential aggregation, reducing the overall work and the likelihood of errors.

Pointer classes divide along two design dimensions, the level of abstraction of the pointerlike behavior and the copy behavior. A pointer class object can point at either a specific type of object or at an interface; copy behavior can be, for example, copied or shared.

Copied-object pointers maintain a one-to-one correspondence between a pointer and an object on the heap. Deleting the pointer deletes the corresponding heap object, thereby freeing the memory and avoiding garbage. Copying the pointer, or using it on the right-hand side of an assignment, copies the heap object.

When a copied-object pointer is used as member data, the C++-generated copy constructor for the class will copy the heap object when it does a memberwise copy of the class object. Such automatic copying eliminates the common error of using a built-in pointer as a data member and forgetting to define a copy constructor. Similarly, the C++-generated destructor and assignment operator will behave appropriately when a copied-object pointer is used instead of a built-in pointer. Moreover, a copied-object pointer member datum initialized in a constructor will recover the heap space if the constructor throws an exception.

Two different copied-object pointers are needed. For creation types, `CopiedObjPtr<T>` gives copied-object pointers to objects of type `T`; for interface base types, `CloneableObjPtr<T>` gives copied-object pointers to objects of types derived from `T`. Both pointer types support controlling referential aggregation but they use different functions to copy objects.

Reference-counted object pointers allow multiple pointers to share (point to) a single heap object. A count, called a *reference count*, is maintained of how many reference-counted object pointers point to a particular heap object, and the heap object is deleted only when the count goes to zero, thus avoiding the dangling reference problem. Copying a reference-counted object pointer does not copy the heap object but increases the count. Destroying a reference-counted object pointer decreases the count. Using reference-counted object pointers as member data eliminates the need to manage manually the destruction of the heap object. We use `Counted` in the name of these pointers. Our `CountedObjPtr<T>` class provides object pointers that are reference counted to ensure responsible referential-sharing aggregation.

The behaviors of copied and reference-counted object pointers are summarized in Table 14.1. Note that copied and reference-counted strategies are not the only possibilities. For example, we could design pointers for *copy-on-write*, where the object is shared until it is modified when it is copied; or we could design pointers that delegate memory management to a garbage collection system. By adding interfaces to pointers, we could create pointers whose memory management strategy was determined at runtime.

	Copied Pointer	*Counted Pointer*
Construct from T*	Point to heap object	Point to heap object with count of one
Copy pointer	Copy heap object	Increase reference count
Destroy pointer	Destroy heap object	Decrease count and destroy heap object if no pointers to it remain
Assign pointer	Destroy left-hand heap object, copy right-hand heap object and point to the copy	Decrease count for left-hand heap object and destroy heap object if no pointers to it remain; increase count for heap object pointed to by right-hand pointer and point to the same heap object
Assign T*	Destroy left-hand heap object, then point to right-hand heap object	Decrease count for left-hand heap object and destroy heap object if no pointers to it remain; point to right-hand heap object with count of one

Table 14.1 Summary of the Behavior of Copied and Reference-Counted Object Pointer Classes. All these classes are parameterized by the type T of the objects pointed to.

14.4 Copied-Object Pointers

Our first copied-object pointer class works with both built-in and class objects but provides no member selection operator (->):

SciEng/CopiedObjPtr.h

```
template<class T>
class CopiedBuiltInPtr {
public:
    CopiedBuiltInPtr(): the_p(0) {}                          // Construct as null pointer
    CopiedBuiltInPtr(T* just_newed):                         // Construct from new pointer
        the_p(just_newed) {}
    ~CopiedBuiltInPtr() { delete the_p; }                    // Discard heap object
    CopiedBuiltInPtr(const CopiedBuiltInPtr<T>&);            // Copy heap object
    CopiedBuiltInPtr<T>& operator=(T*);                      // Delete then assign to new pointer
    CopiedBuiltInPtr<T>&                                     // Copy heap object
        operator=(const CopiedBuiltInPtr<T>&);
    T& operator*() const { return *the_p; }                  // Access object
    Boolean isNull() const { return the_p == 0; }
    T* releaseControl();                                     // Given up lifetime control; no delete
```

```
    friend
    Boolean operator==(const CopiedBuiltInPtr<T>& lhs, const CopiedBuiltInPtr<T>& rhs) {
        return lhs.the_p == rhs.the_p;
    }
    friend
    Boolean operator!=(const CopiedBuiltInPtr<T>& lhs, const CopiedBuiltInPtr<T>& rhs) {
        return lhs.the_p != rhs.the_p;
    }
protected:
    T* the_p;
};
```

The class provides the ability to access the object it points to by overloading the indirection operator (*), the ability to test for a null pointer with isNull(), and pointer equality testing with global equality test operators. Neither pointer arithmetic nor pointer comparison (other than equality and inequality) is provided because this class is designed to work with individual objects, not arrays of objects.

Internally, these objects are just pointers to T objects. However, all of their creation and destruction behavior is completely different from built-in pointers: A CopiedBuiltInPtr<T> controls exactly one T object.

The one-to-one connection between pointer and heap object is maintained through the controlling behavior—constructing, deleting, copying, and assigning—implemented by the constructors, destructor, and assignment operators. The mechanisms implemented are illustrated in Fig. 14.3.

As the inline member functions show, when a copied-object pointer is created from a T*, the pointer is stored in the built-in pointer member datum the_p. The destructor simply deletes the heap object pointed to. Copying a CopiedObjPtr<T> must cause the corresponding heap object to also be copied, maintaining the one-to-one correspondence between a copied-object pointer and the heap object:

SciEng/CopiedObjPtr.c

```
template<class T>
CopiedBuiltInPtr<T>::CopiedBuiltInPtr(const CopiedBuiltInPtr<T>& aCP) :
    the_p(aCP.isNull() ? 0 : new T(*aCP.the_p)) {
}
```

The copy constructor creates a new object on the heap using T's copy constructor, invoked by new.

Assigning a T* (assumed to be the result of new) to a copied-object pointer causes the old heap object to be deleted and the copied-object pointer to point at the new heap object:

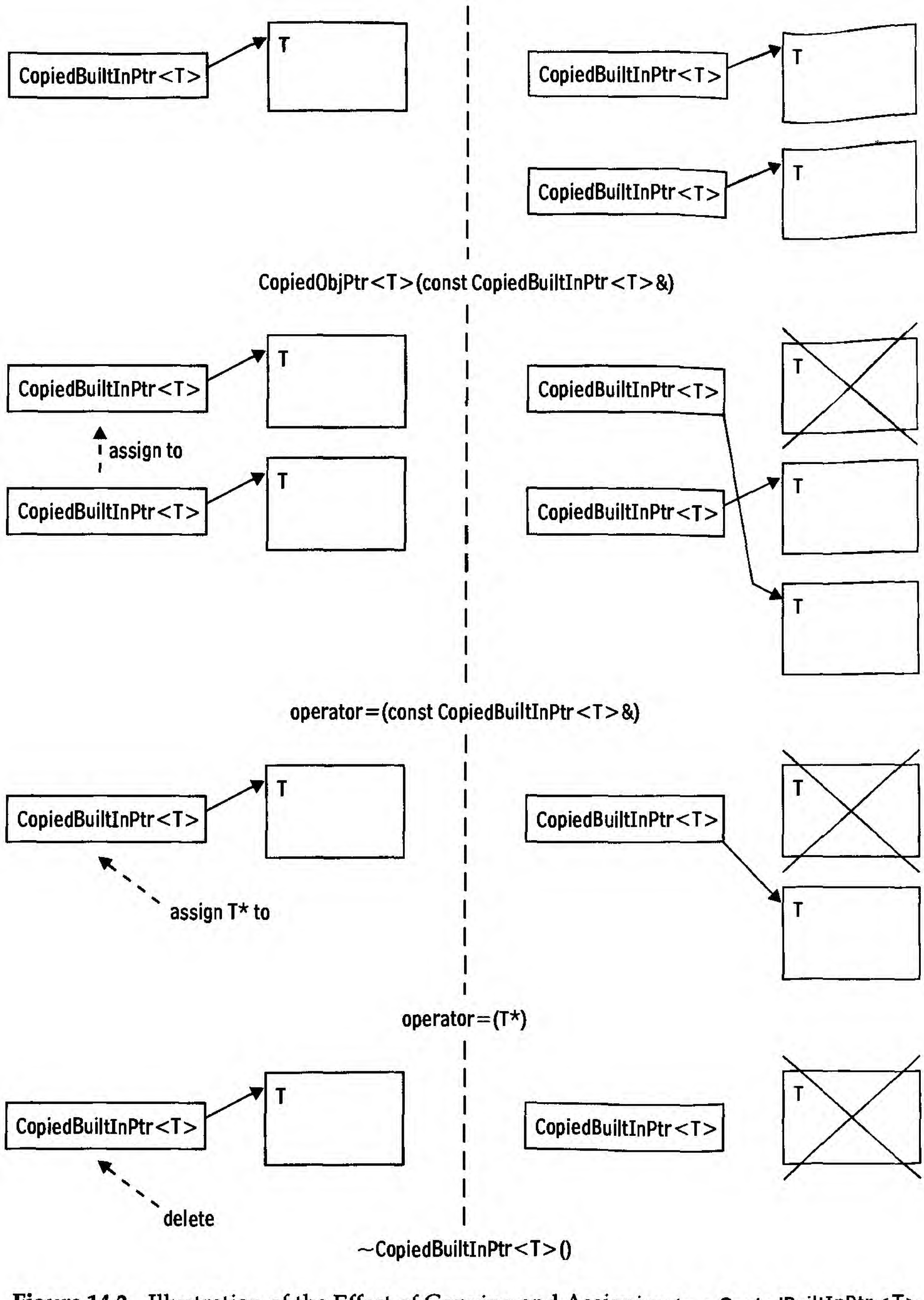


Figure 14.3 Illustration of the Effect of Copying and Assigning to a CopiedBuiltInPtr<T> The two sides illustrate before (left) and after (right).

SciEng/CopiedObjPtr.c

```
template<class T>
CopiedBuiltInPtr<T>& CopiedBuiltInPtr<T>::operator=(T* just_newed) {
    delete the_p;              // Discard lhs object.
    the_p = just_newed;        // Point to just_newed object.
    return *this;
}
```

Similarly, assigning one copied-object pointer to another deletes the heap object pointed to by the left operand, makes a copy of the heap object pointed to by the right operand, and points to the copy:

SciEng/CopiedObjPtr.c

```
template<class T>
CopiedBuiltInPtr<T>& CopiedBuiltInPtr<T>::operator=(const CopiedBuiltInPtr<T>& rhs) {
    if (the_p != rhs.the_p) {
        delete the_p;
        the_p = rhs.isNull() ? 0 : new T(*rhs.the_p);
    }
    return *this;
}
```

The test guards against the possibility of a copied-object pointer being assigned to itself: Without the test, the (one) heap object would be deleted and then copied.

The releaseControl() member function provides safe access to the heap object. Control is released by setting the the_p pointer to null, preventing multiple pointers to the same object without clear lifetime control:

SciEng/CopiedObjPtr.c

```
template<class T>
T* CopiedBuiltInPtr<T>::releaseControl() {
    T* save_p = the_p;
    the_p = 0;
    return save_p;
}
```

The copy constructor and assignment operator test for and ignore a null pointer, and the C++ delete operator ignores null pointers. Therefore copying, assigning, or deleting a copied-object pointer after it has released control has no effect, as desired.

CopiedObjPtr<T> adds the member selection operator to the functions it inherits from CopiedBuiltInPtr<T>:

SciEng/CopiedObjPtr.h

```
template<class T>
class CopiedObjPtr :
    public CopiedBuiltInPtr<T> {
public:
    CopiedObjPtr(const CopiedObjPtr<T>& p) : CopiedBuiltInPtr<T>(p) {}
    CopiedObjPtr() : CopiedBuiltInPtr<T>() {}
    CopiedObjPtr(T* just_newed) : CopiedBuiltInPtr<T>(just_newed) {}
    CopiedObjPtr<T>& operator=(T* rhs) {
        CopiedBuiltInPtr<T>::operator=(rhs);
        return *this;
    }
    CopiedObjPtr<T>& operator=(const CopiedObjPtr<T>& rhs) {
        CopiedBuiltInPtr<T>::operator=(rhs);
        return *this;
    }
    T* operator->() const { return the_p; }
};
```

The member selection operator -> must return either a built-in pointer to a class object or an object of another pointerlike class with a -> operator; a built-in pointer to a built-in type object cannot be returned. C++ cascades all of the in-
§13.4.6 direction operator functions until one of them returns a built-in pointer leading to the final indirection. `CopiedObjPtr<T>` also provides constructors and assignment operators, which cannot be inherited, with the work done by the corresponding base class functions.

A `CopiedBuiltInPtr<T>` or `CopiedObjPtr<T>` object is constructed from a `T*`. This must be a pointer to a single object (not to an array of objects) that is returned by `new`.

- Construct a copied-object pointer directly from the result of the `new` expression that allocates the heap object.

This style allows the user to specify arguments to the constructor for the heap object and decouples `CopiedObjPtr<T>` from the specifics of any one class. This style also avoids the danger of constructing a `CopiedObjPtr<T>` from a `T*` used elsewhere. (When the `CopiedObjPtr<T>` is destroyed, the heap object to which it points is deleted, causing other pointers to that object to become dangling references.)

Using `CopiedBuiltInPtr<T>`, we can rewrite the example class `Leaker` (page 423) like this:

ch14/exceptionNoLeak.C

```
class NonLeaker {
public:
    NonLeaker(Subscript size) : p(new double(0)), array(size) {}
private:
    CopiedBuiltInPtr<double> p;
    ConcreteFormedArray1d<int> array;
};

void f() {
    // ...
    NonLeaker l(-3);
}
```

We no longer need to provide a copy constructor, a destructor, or a copy assignment operator: The necessary behavior is supplied by the copied-object pointer's copy constructor, destructor, and copy assignment operator. Moreover, there is no memory leak when the exception is thrown while constructing the array member datum: Although destructors for partially constructed objects are not run when an exception is thrown, the destructors for fully constructed member data are run. The copied-object pointer was fully constructed: Its destructor is run and the dynamically allocated memory freed properly. ARM

More substantive examples of using copied-object pointers appear in later chapters. Look at the constructor for the class LapackRect<T> on page 461. The copied-object pointer ap is initialized directly from the new expression that creates the heap object. Similarly, the copied-object pointer fmat_p in the constructor for FactoredLapackRect<T> on page 463 is set by an object directly from new.

14.5 Counted-Object Pointers

A *reference-counted object pointer class* allows multiple pointers to point at a single heap object but ensures that the heap object is deleted when all pointers to it are gone. Our version maintains an explicit count of pointers that refer to the heap object, so it is called a *reference-counted pointer class*.

Counted pointers are useful for implementing sharing-referential aggregation. Like the copying pointers from the preceding section, reference-counted pointer member data allow C++-generated copy constructors, assignment operators, and destructors to work for component objects held by pointers. Unlike CopiedObjPtr<T>, CountedObjPtr<T> does not maintain a one-to-one relation between pointers and objects, but a many-to-one relation. See Fig. 14.4.

As with our copied-object pointers, reference-counted object pointers provide basic pointerlike behavior by overloading the indirection operator (*), the ability

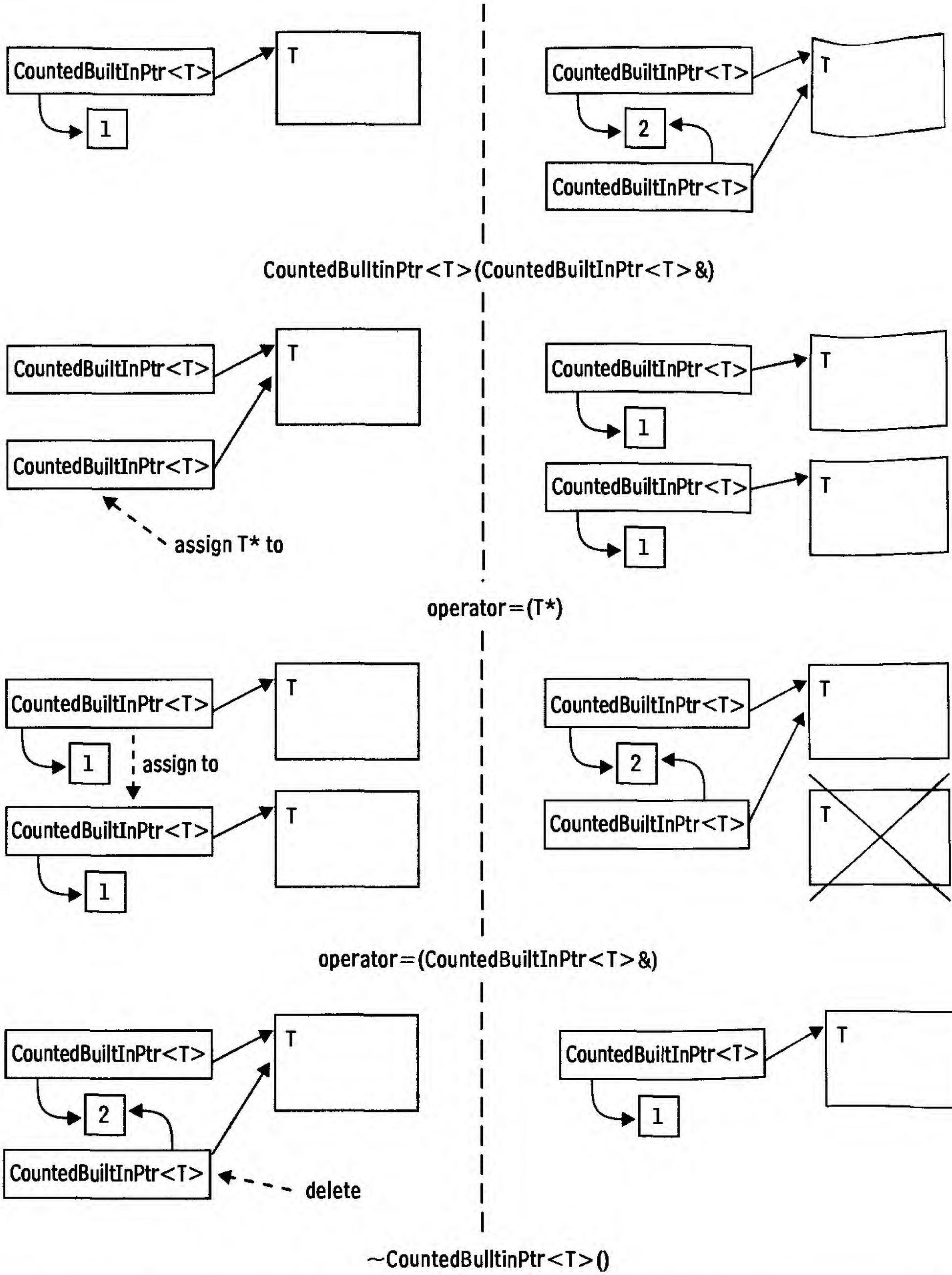


Figure 14.4 Illustration of the Effect of Constructing, Copying, Assigning to, and Destroying a CountedBuiltInPtr<T>. The two sides illustrate before (left) and after (right) the specified function is executed.

to test for a null pointer with isNull(), and pointer equality testing with global equality test operators:

SciEng/CountedObjPtr.h

```
template<class T>
class CountedBuiltInPtr {
public:
  CountedBuiltInPtr() : the_p(0) {}                    // Construct as null pointer
  CountedBuiltInPtr(T* just_newed) :                   // Construct pointing at heap object
    the_p(just_newed) {}
  CountedBuiltInPtr<T>&                                // Adjust counts
    operator=(const CountedBuiltInPtr<T>&);
  CountedBuiltInPtr<T>& operator=(T*);                 // Decrease lhs count
  ~CountedBuiltInPtr();                                // Decrease count, destroy if 0
  Boolean unique() const;                              // Is count one ?

  T& operator*() const { return *the_p; }              // Access object
  Boolean isNull() const { return the_p == 0; }

  friend
  Boolean operator==(const CountedBuiltInPtr<T>& lhs, const CountedBuiltInPtr<T>& rhs) {
    return lhs.the_p == rhs.the_p;
  }
  friend
  Boolean operator!=(const CountedBuiltInPtr<T>& lhs, const CountedBuiltInPtr<T>& rhs) {
    return lhs.the_p != rhs.the_p;
  }
protected:
  T* the_p;
private:
  ReferenceCount refCount;     // Number of pointers to heap object
};
```

Neither pointer arithmetic nor pointer comparison (other than equality and inequality) is provided because this class is designed to work with individual objects, not arrays of objects.

Internally, these objects have a pointer to a T object and a ReferenceCount data member, which holds the number of references to the object referenced by this pointer object. The ReferenceCount member does much of the work, so we describe it before proceeding with the members of CountedBuiltInPtr<T>.

A ReferenceCount object that is created by copying another ReferenceCount object must be forever coupled to the original object: All copies of a ReferenceCount object

have the same count. From among several plausible implementations, we have chosen to have all copies point to the same counter, allocated in the heap and managed using C++'s native pointer facilities:

SciEng/ReferenceCount.h

```
class ReferenceCount {
public:
    ReferenceCount();                              // Create with count of 1
    ReferenceCount(const ReferenceCount&); // Copy and increment count
    ReferenceCount&                                // Assign; decrement lhs count, increment rhs
        operator=(const ReferenceCount&);
    ~ReferenceCount();                             // Decrement count; delete if 0
    Boolean unique() const;                        // True if count is 1
private:
    unsigned int* p_refcnt;                        // Pointer to actual count
    void decrement();                              // Decrement count; delete if 0
};
```

(We would like to use a CountedObjPtr<T>, but we can't because ReferenceCount is part of its implementation!) All of the action for a ReferenceCount object occurs when it is created, copied, assigned, or destroyed. The only other function provided is to ask whether the reference is unique (i.e. whether the count is one).

When a ReferenceCount is created, the unsigned int member is allocated in the heap with an initial value of one:

SciEng/ReferenceCount.h

```
inline ReferenceCount::ReferenceCount() : p_refcnt(new unsigned int(1)) {}
```

The copy constructor copies the pointer to this counter and increments it by one:

SciEng/ReferenceCount.h

```
inline ReferenceCount::ReferenceCount(const ReferenceCount& anRC) :
    p_refcnt(anRC.p_refcnt) { ++*p_refcnt; }
```

Therefore after a copy constructor call we have two ReferenceCount objects, each containing a pointer to the same int; the value stored in the int is 2.

The destructor decrements the count by one and deletes the unsigned int from the heap if the count goes to zero:

SciEng/ReferenceCount.h

```
inline void ReferenceCount::decrement() {
    if (unique()) delete p_refcnt;
    else --*p_refcnt;
}

inline ReferenceCount::~ReferenceCount() { decrement(); }
```

For example, if one of the two ReferenceCount objects in the preceding paragraph were deleted, the remaining object would point to an int with value 1.

Assignment adds to the pool of objects represented by the right side of the = sign and removes from the pool of objects represented by the left side. Then the object on the left side points to the same int as the right side:

SciEng/ReferenceCount.h

```
inline ReferenceCount& ReferenceCount::operator=(const ReferenceCount& rhs) {
    ++*rhs.p_refcnt;
    decrement();
    p_refcnt = rhs.p_refcnt;
    return *this;
}
```

Incrementing the right-hand count before decrementing the left-hand count protects against self-assignment problems.

With ReferenceCount in hand, we can implement reference-counted object pointers. As shown in the inline function definitions, a reference-counted object pointer is constructed from a built-in pointer by setting the internal pointer and initializing the reference count via the default constructor for ReferenceCount. As with copied-object pointers,

- Construct a reference-counted object pointer directly from the result of the new expression that allocates the heap object.

This is illustrated in the example in Section 14.6.

A reference-counted object pointer is copied by copying the internal pointer and the reference count. Since this is the behavior provided by the C++-generated copy constructor, we don't need to supply our own. The copy constructor for the reference count increments the count. Both assignment operators set the internal pointer to point at a new heap object and adjust the reference count:

SciEng/CountedObjPtr.c

```
template<class T>
CountedBuiltInPtr<T>&
CountedBuiltInPtr<T>::operator=(const CountedBuiltInPtr<T>& rhs) {
    if (the_p != rhs.the_p) {
        if (unique()) delete the_p;
        the_p = rhs.the_p;
        refCount = rhs.refCount;
    }
    return *this;
}
```

```
template<class T>
CountedBuiltInPtr<T>& CountedBuiltInPtr<T>::operator=(T* just_newed) {
    if (unique()) delete the_p;
    the_p = just_newed;
    refCount = ReferenceCount();
    return *this;
}
```

Both operators delete the heap object pointed to by the left-hand side if necessary and then update the internal pointer. The copy assignment operator copies the right-hand reference count, which increments it, while the other assignment operator assigns a newly created reference count. The destructor deletes the heap object if the reference-counted object pointer being destroyed points at it uniquely:

SciEng/CountedObjPtr.c

```
template<class T>
CountedBuiltInPtr<T>::~CountedBuiltInPtr() {
    if (refCount.unique()) delete the_p;
}
```

For pointing to class objects, the member selection operator is added by `CountedObjPtr<T>`:

SciEng/CountedObjPtr.h

```
template<class T>
class CountedObjPtr :
    public CountedBuiltInPtr<T> {
public:
    CountedObjPtr() {}
    CountedObjPtr(T* just_newed) : CountedBuiltInPtr<T>(just_newed) {}
    CountedObjPtr<T>& operator=(T* rhs) {
        CountedBuiltInPtr<T>::operator=(rhs);
        return *this;
    }
    CountedObjPtr<T>& operator=(const CountedObjPtr<T>& rhs) {
        CountedBuiltInPtr<T>::operator=(rhs);
        return *this;
    }
    T* operator->() const { return the_p; }
};
```

The constructors and assignment operators are necessary because they are not inherited.

14.6 Using Counted-Object Pointers

Since object sharing is more involved than simple containing aggregation, we need a larger example to illustrate reference-counted object pointers. Reference-counted object pointers help when we need both object sharing and dynamic allocation. Sharing objects that are fixed at compile time can be accomplished through the use of static member data.

For our example we return to the finite element mesh program discussed in Chapter 8, extending the problem to make it more dynamic. We want to allow specific elements in the mesh to be deleted (e.g., because the physical entity being analyzed by the finite element program has been interactively made smaller). In real meshers, many such dynamic operations on the mesh must be provided to allow users to tune the mesh.

Recall that the mesh program read data describing a mesh in terms of nodes (points in space) and elements (regions bounded by edges drawn between nodes). Our final version defined classes for Mesh, Node, Element, iterators NodesOfElement and ElementsOfMesh, as well as some auxiliary classes.

The solution in Chapter 8 is inefficient if we want to modify the mesh because the elements and nodes are stored in arrays. To delete an element requires moving all of the other elements, a large task for a large mesh. Moreover, deleting an element might result in some of the nodes being left out. For example, if we trim the mesh of Fig. 8.3 to the bounding box (8.5,5.5) shown in Fig. 14.5, some of the nodes will not be referenced by any element and could be deleted. With arrays we would have to sweep through all elements and mark the nodes still accessed before we could be sure which nodes are unused and could be deleted.

A more dynamic solution uses a structure based on linked lists and pointers. We use the List<T> class template from page 170 for our linked lists. Our Node objects are still just Points, as in Section 8.5. Our new version of the Mesh class looks like this:

ch14/tCounted.C

```
class Mesh {
public:
    friend ElementsOfMesh;      // Iterator over elements of a mesh
    friend NodesOfMesh;         // Iterator over nodes of a mesh
    friend istream& operator>>(istream&, Mesh&);

    Boolean checkElementAngles(Number angle_threshold) const;
    void remove(const List<CountedObjPtr<Element> >& trim);
private:
    List<CountedObjPtr<Element> > element_table;
};
```

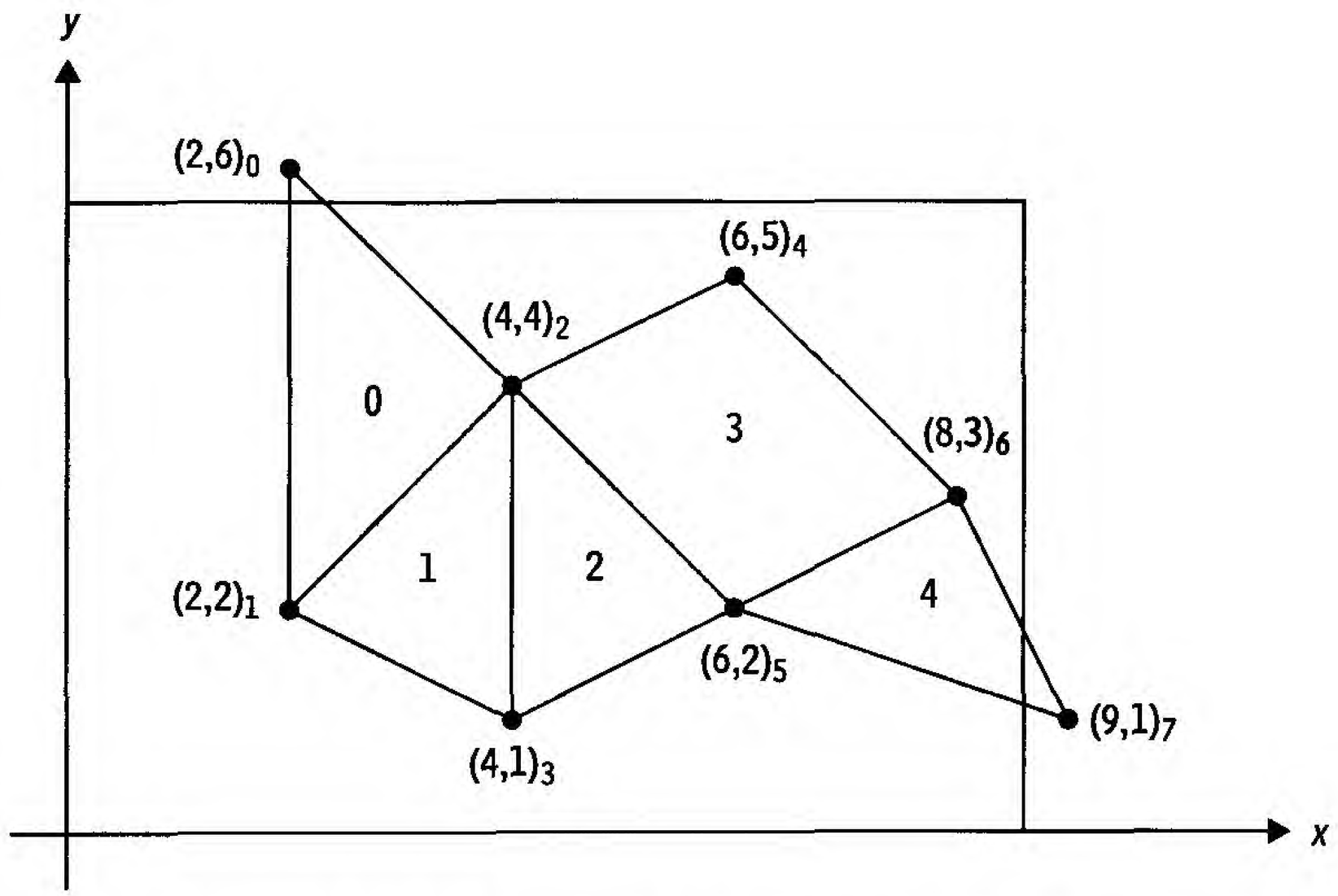


Figure 14.5 The Mesh of Fig. 8.1 with a Bounding Box Indicating Elements to be Eliminated.

We have added a member function remove() to remove elements specified in the argument; we describe this on page 440. The two arrays in the original code, one for Nodes and one for Elements, have been replaced by a single linked list; the array also implicitly indexed the Nodes as they were read in so they could be connected to the Elements, a job now localized in NodeReader.

The elements themselves are almost the same as in Section 8.5:

ch14/tCounted.C

```
class Element {
public:
    friend NodesOfElement;    // Iterator over nodes of an element

    friend void operator>>(NodeReader& reader, Element& e);
    friend ostream& operator<<(ostream& os, const Element& e);
    Number maxAngle() const;
private:
    SimpleArray< CountedObjPtr<Node> > node_ptrs;
};

inline Boolean operator==(const Element& lhs, const Element& rhs) {
    return &lhs == &rhs;
}
```

```
inline Boolean operator!=(const Element& lhs, const Element& rhs) {
    return !(lhs == rhs);
}
```

The built-in pointers to Node objects have been replaced by CountedObjPtr<Node>, and equality tests have been added as required by List::remove() (Exercise 6.9). Since elements are not copied, we consider two elements to be equal if they are the same object. Deleting an Element from the Mesh will cause the element's array to be destroyed, which in turn causes destructors for all of the CountedObjPtr<Node> objects to be run. The reference counts for the nodes will be reduced, and those that are no longer referenced will be deleted. Thus memory management of the Element aggregate components is done by smart pointers.

Changing from array-based storage to list-based storage also requires changing the iterator over elements in the mesh:

ch14/tCounted.C

```
class ElementsOfMesh :
    public ListIterator<CountedObjPtr<Element> > {
public:
    ElementsOfMesh(const Mesh& m) :
        ListIterator<CountedObjPtr<Element> >(m.element_table) { }
};
```

This class simply specializes ListIterator<T> from page 170. Notice that ListIterator<T> uses sharing if T is a pointer type. By using CountedObjPtr<Element> for T, rather than Element*, this sharing is reference counted. The NodesOfElement class need not be changed.

With the mesh, elements, and nodes modified, we fix up the auxiliary classes and add the function to remove elements. The NodeReader gets an internal array that can be indexed as elements are read in:

ch14/tCounted.C

```
class NodeReader {
public:
    NodeReader(int num_nodes, istream& instream);
    int getSize();
    CountedObjPtr<Node> getNode();
private:
    SimpleArray< CountedObjPtr<Node> > node_table;
    istream& in;
};
```

It is no longer coupled to Mesh since it does not need to share the Mesh member data: Sharing occurs through copies of CountedObjPtr<Node>. The NodeReader constructor calls new for each Node it reads:

ch14/tCounted.C

```
NodeReader::NodeReader(int num_nodes, istream& instream) :
    node_table(num_nodes), in(instream) {
    // Read nodes
    for (int nodeNum = 0; nodeNum < num_nodes; nodeNum++) {
        node_table[nodeNum] = new Node;
        instream >> *(node_table[nodeNum]);
    }
}
```

Later, when the NodeReader is called to load an Element like this

ch14/tCounted.C

```
void operator>>(NodeReader& reader, Element& e) {
    int nNodesInElement = reader.getSize();
    e.node_ptrs.setSize(nNodesInElement);
    for (int i = 0; i < nNodesInElement; i++) {
        e.node_ptrs[i] = reader.getNode();
    }
}
```

the NodeReader's getNode() member returns a CountedObjPtr<Node>:

ch14/tCounted.C

```
CountedObjPtr<Node> NodeReader::getNode() {
    int nodeNum;
    in >> nodeNum;
    return node_table[nodeNum];
}
```

The NodeReader can be deleted after the data are read, and the reference counting ensures that the Elements still refer to valid Node objects.

The remove() member of Mesh accepts a list of reference-counted object pointers to elements and removes each one from the element table:

ch14/tCounted.C

```
void Mesh::remove(const List<CountedObjPtr<Element> >& trim) {
    for (ListIterator<CountedObjPtr<Element> > i(trim); i.more(); i.advance()) {
        element_table.remove(i.current());
    }
}
```

For example, the following function tests for any part of an element outside of the box bounded by the origin and the argument bound and removes them:

ch14/tCounted.C

```
void trimTo(Mesh& m, const Point& bound) {
    List<CountedObjPtr<Element> > trim;    // Accumulator for out of bound elements
    for (ElementsOfMesh elts(m); elts.more(); elts.advance()) {
        for (NodesOfElement nodes(*elts.current()); nodes.more(); nodes.advance()) {
            if ( (nodes.current().x() > bound.x()) || (nodes.current().y() > bound.y())) {
                trim.add(elts.current());
            }
        }
    }
    m.remove(trim);    // Delete the out of bounds elements.
}
```

Not only will the Element objects be properly memory managed, but any Node objects no longer referenced by an Element will be reclaimed.

We motivated the conversion from an array-based solution to a list-based solution by efficiency considerations. Whether a list-based solution is really faster than an array-based solution depends on many factors. Array-based solutions require lots of copying, but list-based solutions (with or without reference counting) require more memory. Reference counting also adds both space and time costs (see Exercise 14.8). Which solution provides the best performance depends on characteristics of the mesh such as the average number of nodes per element and the average number of elements sharing a node. Other considerations also arise in practice. For example, sophisticated data structures that require pointer-based implementations can reduce the cost of various mesh queries (e.g., given an element, find all adjacent elements). If an application does many such queries, such data structures are advantageous.

So the point of this example is not that list-based meshes are inherently better than array-based meshes, but rather that smart pointers can be useful in implementing list-based meshes and other pointer-based data structures. Whenever shared aggregation is used, code using reference-counted pointers will likely be more obvious and easier to maintain than equivalent code using built-in pointers.

14.7 Interface Pointer Classes

Pointers used for aggregation with interface categories, as described in Chapter 9, must be able to point to an object that is an instance of any member of the interface category. The CountedObjPtr<T> pointers from the preceding section can do this, but if we try to use CopiedObjPtr<T> pointers to create a controlling-referential aggregate knowing only the base type, we fail. The CopiedObjPtr<T> pointers copy objects by invoking the new operator and T's copy constructor. A built-in pointer

to an interface base class cannot be used to copy the heap object because we do not know which copy constructor to call.

For example, suppose we are writing part of an interactive program to represent molecules composed of atoms. Ignoring all of the details except the memory management, we want to represent a Molecule without knowing until the program runs which atoms will be in which molecules. Since the properties of each hydrogen or oxygen atom are constant, we represent these as classes. Each instance of class Hydrogen will represent one hydrogen atom. By deriving all of the different types of atoms from Atom, a Molecule can hold a collection of Atom* to represent a molecule.

Users of our program will want to have several water molecules. They might create a water molecule by combining, at runtime, two Hydrogen objects with one Oxygen object. Then they would want to copy the resulting Molecule to create duplicates. Our system must support this.

To support such copies, we want referential-controlling aggregation, but the Molecule class alone does not have enough information about the objects it contains to copy them. Given an Atom*, should we call the Oxygen or Hydrogen copy constructors? We need what is called a *virtual constructor*, a constructor that is part of the interface specified in the interface base class. C++ does not provide virtual constructors.

We can, however, add virtual member functions that act like copy constructors. For example, we can add a function clone() to the Atom interface like this:

ch14/tMolecule.C

```
class Atom {
public:
    virtual String       name()         const = 0;
    virtual unsigned int atomicNumber() const = 0;
    virtual Atom*        clone()        const = 0;
    virtual              ~Atom() {}
};
```

In derived classes, we define this virtual function to call the copy constructor and return a pointer to the heap:

ch14/tMolecule.C

```
class Hydrogen :
    public virtual Atom {
public:
    virtual String       name()         const { return "Hydrogen"; }
    virtual unsigned int atomicNumber() const { return 1; }
    virtual Hydrogen*    clone()        const { return new Hydrogen(*this); }
};
```

Notice that the return type for the virtual clone() function here is a pointer to the derived class, but C++ will allow this much difference and still consider the base class virtual function to be matched. (See Notes and Comments 14.5.)

- Return a pointer to the derived class in clone() functions.

With this interface, we can duplicate a list of Atom pointers:

ch14/tMolecule.C

```
void duplicate(       ConcreteFormedArray1d<Atom*>& lhs,
                const ConcreteFormedArray1d<Atom*>& rhs) {
    ConcreteFormedArray1d<Atom*>::IteratorType lhs_i(lhs);
    ConcreteFormedArray1d<Atom*>::BrowserType rhs_i(rhs);
    for(; lhs_i.more(); lhs_i.advance(), rhs_i.advance()) {
        lhs_i.current() = rhs_i.current()->clone();
    }
}
```

Notice that we cannot rely on the copy assignment operator for ConcreteArray1d<Atom*>, a flaw we fix later in this chapter.

This approach to virtual copy constructors is prone to one serious error:

- In systems using virtual clone() functions, be sure to override clone() in every derived class.

Only classes derived from abstract interface base classes will give compile errors if you fail to override clone(). Failing to override the function will lead to subtle bugs when the wrong object is copied.

There is no comparable problem for destructors, since C++ provides virtual destructors. We recommend supplying one in every interface base class, as described in Section 9.3.2.

Once we have introduced the clone() functions to the interface, we are still left with manually overriding the default sharing aggregation of built-in pointers. This can be accomplished automatically using another smart pointer class:

SciEng/CloneableObjPtr.h

```
template<class T>
class CloneableObjPtr {
public:
    CloneableObjPtr(): the_p(0) {}                      // Construct as null pointer
    CloneableObjPtr(T* just_newed):                     // Construct from new pointer
        the_p(just_newed) {}
    ~CloneableObjPtr() { delete the_p; }                // Discard heap object
    CloneableObjPtr(const CloneableObjPtr<T>&);         // Copy heap object
    CloneableObjPtr(const T& obj) :                     // Copy object
        the_p(obj.clone()) {}
```

```
    CloneableObjPtr<T> & operator=(T*);          // Delete then assign to new pointer.
    CloneableObjPtr<T> &                         // Copy heap object
      operator=(const CloneableObjPtr<T> &);
    T& operator*() const { return *the_p; }      // Access object
    Boolean isNull() const { return the_p == 0; }
    T* releaseControl();                         // Given up lifetime control; no delete.
    T* operator->() const { return the_p; }
    friend
    Boolean operator==(const CloneableObjPtr<T>& lhs, const CloneableObjPtr<T>& rhs) {
      return lhs.the_p == rhs.the_p;
    }
    friend
    Boolean operator!=(const CloneableObjPtr<T>& lhs, const CloneableObjPtr<T>& rhs) {
      return lhs.the_p != rhs.the_p;
    }
protected:
    T * the_p;
};
```

This class is similar to CopiedObjPtr<T>. The main difference is the use of clone() instead of copy constructors whenever the referenced object is copied. For example, the copy constructor is

SciEng/CloneableObjPtr.c

```
template<class T>
CloneableObjPtr<T>::CloneableObjPtr(const CloneableObjPtr<T>& aCP) :
    the_p(aCP.isNull() ? 0 : aCP.the_p->clone()) {
}
```

The other members are obvious and not shown.

Using this class, we can implement a simple Molecule class without concern for the details of controlling aggregation:

ch14/tMolecule.C

```
class Molecule {
public:
    Molecule(Subscript num_atoms) : atoms(num_atoms) {}
    Subscript numAtoms() const { return atoms.numElts(); }
    const CloneableObjPtr<Atom>& operator()(Subscript i) const { return atoms(i); }
          CloneableObjPtr<Atom>& operator()(Subscript i)       { return atoms(i); }
    void readAtoms();

private:
    ConcreteFormedArray1d< CloneableObjPtr<Atom> > atoms;
};
```

Copy constructors and assignment operators need not be coded.

As with the CopiedObjPtr<T> and CountedObjPtr<T>, we create the CloneableObjPtr<T> from pointers returned by new:

ch14/tMolecule.C

```
void Molecule::readAtoms() {
    Molecule& m = *this;
    for (Subscript i = 0; i < m.numAtoms(); i++) {
        cout << "Enter H for hydrogen, O for oxygen:";
        String c;
        if (cin >> c) {
            if      (c == "Hydrogen") m(i) = new Hydrogen;
            else if (c =="Oxygen")    m(i) = new Oxygen;
        }
        else throw BadMoleculeInput();
    }
}
```

In addition to keeping the pointer class ignorant of the constructor arguments, this approach allows the pointer to be created only for the base class: The CloneableObjPtr<Atom> in the Molecule is created either from the Hydrogen pointer or from the Oxygen pointer as required. No smart pointer objects for either derived class are needed.

One problem with CloneableObjPtr<T> is not evident in this example: Whereas a built-in pointer to a base class can be initialized with a built-in pointer to a derived class, we cannot initialize a CloneableObjPtr<Base> from a CloneableObjPtr<Derived> even when Derived is derived from Base. The two smart pointer classes are not related. This becomes important when we use base class composition as described in Chapter 10 and the Derived class is in fact another layer of interface. We could implement conversions to support this, but in large systems the maintenance of such conversions is troublesome. In practice we call the clone() function on the smart pointer manually in these cases.

14.8 Summary

Pointers are simultaneously powerful and dangerous. We traced some of the dangers to the lack of different kinds of pointer types and to the default *irresponsible sharing* of aggregated built-in pointers. By introducing new pointer types, the compiler is told how we intend for our pointers to be used, and it can then check that we use these pointers only in those ways. We introduced pointer types for aggregation that support controlling aggregation and responsible, shared aggregation through reference counting. We showed the most common way of supporting copies of objects known only through interfaces and provided a pointer type for controlling aggregation of such objects.

In our experience, the CopiedObjPtr<T> pointers work well and manual coding to support controlling-referential aggregation is rarely justified. The CountedObjPtr<T> also works well, but manual coding to support sharing-referential aggregation can produce faster and smaller programs if the memory management is successful. The improvements may or may not be significant compared with the effort.

The CloneableObjPtr<T> suffers from the potential error of failing to override the clone() member and from the lack of conversions that parallel the conversions available for built-in pointers. Otherwise they work well: Their performance is rarely an issue.

14.9 Notes and Comments

14.1 We would be able to eliminate the BuiltInPtr<T> layer of classes if either member functions of class templates were not expanded into template class member function definitions unless actually used, or if operator->() could be defined to return a pointer to a built-in object (e.g., float*). Currently an operator->() cannot return a float* because members cannot be selected for built-in objects: For float* f, the expression f->x cannot be legal for any x. However, the language could be defined to allow the operator->() definition while making its actual application illegal. Thus the definition of a programmer-defined pointer class could include operator->() as long as it was not called for a built-in type. See also Exercise 14.4.

14.2 Garbage collection is an alternative to reference counting. The negotiation between the pointers referencing a shared object is delayed until memory runs out, and then all pointers that might point at memory are checked and the memory that is not referenced directly or indirectly is reclaimed. Reference counting is easier to implement than garbage collection and distributes the runtime overhead more evenly than classical garbage collection algorithms do. On the other hand, the total overhead of reference counting over the entire execution of a program can be higher than that of garbage collection, space is consumed for the reference counts, and linked data structures having circularities are not reclaimed. Various approaches to solving these problems have been devised; see [24]. Other C++ implementations of reference counting appear in [51, Section 14.4.1] and [27, Section 3.5].

14.3 Unfortunately there is no portable way to ensure that CopiedObjPtr<T> objects are initialized with pointers to the heap rather than pointers to the stack or pointer to arrays on the heap. If the address-of operator (&) and built-in arrays are avoided, stack pointers and arrays on the heap become rare and this flaw does not become a practical problem.

14.4 The design of smart pointers in C++ is discussed by Edelson [43], who takes the view that the behavior of smart pointers should be a superset of the behavior of built-

in pointers. He shows that this goal is not achievable in C++ and evaluates several approaches to smart pointers. The principal difficulties are relating smart pointers to the class DAG of the objects they point to and handling const objects (see Exercise 14.6). We agree that C++ should allow smart pointers to do any and all things that built-in pointers can do, but we would then use this ability to create a family of restricted pointers suitable for particular purposes.

14.5 As discussed in Notes and Comments 9.10, the ability to have a virtual function return a pointer to a derived type when the interface specifies a pointer to base as the return type is a recent change in C++. If your compiler does not yet allow this, have the clone() function return a pointer to the base. This will work fine unless the derived class is derived from two or more interfaces, each of which requires a clone() function.

14.6 Smart reference-counted pointers are not completely safe as C++ is currently defined, as observed by Kennedy [64]. The problem is related to C++'s rule for when temporary objects can be destroyed. Since most current C++ compilers do not destroy temporaries immediately after they become unusable, this is not a problem in practice. Moreover, the ANSI C++ committee is likely to change the rules to something like the behavior implemented in current compilers. Therefore we have not discussed this safety issue. See [64] or [43] for details. Also see Notes and Comments 18.6.

14.7 We could have derived CloneableObjPtr<T> from CopiedObjPtr<T>, but (with our compiler at least) generation of CloneableObjPtr<Atom> causes all of the members of the base class CopiedObjPtr<Atom> to be generated, including the base class copy constructor that cannot compile since it calls a copy constructor for type Atom. We could have defined a template specialization for this copy constructor, but we would have to do this for every use of CloneableObjPtr<T>. Therefore we chose code duplication in this case.

14.10 Exercises

14.1 Why is it incorrect to construct a CopiedObjPtr<T> from a pointer to an array of T objects?

14.2 What is the output from the following code?

ch14/ptrDemo.h

```
class B :
    public TraceLifetime {
public:
    B(int a) : b(a), TraceLifetime("B") {}
    B(const B& aB) : b(aB.b), TraceLifetime("B-from-B") {}
    int val() const { return b; }
```

```
private:
    int b;
};
class A {
public:
    A(B* p):pb(p) {}
    A& operator=(B* rhs) { pb = rhs; return *this; }
    int val() const { return pb->val(); }
private:
    CopiedObjPtr<B> pb;
};

int main() {
    B* b1 = new B(1);
    A a1(b1);
    A a2(a1);
    cout << a1.val() << " " << a2.val() << endl;
    a1 = new B(2);
    cout << a1.val() << endl;
    a2 = a1;
    cout << a2.val() << endl;
    a2 = a2;
    cout << a2.val() << endl;
    return 0;
}
```

Explain.

14.3 What would be the output from the code of Exercise 14.2 if the `CopiedObjPtr<T>` member was changed to a `CountedObjPtr<T>`? Explain.

14.4 Reimplement `CopiedObjPtr<T>` and `CountedObjPtr<T>` without any of the `BuiltIn` classes as if no built-in objects would be pointed to. Then define template specializations for each built-in type to ensure that all objects can be manipulated through consistent pointer classes.

14.5 The pointer classes described in Sections 14.4 and 14.7 cannot point to a const object. Explain why.

14.6 Design and implement a family of pointer classes that can point to both const and non-const objects. *Hint:* Review the class DAG for the arrays of Chapter 13.

14.7 Design and implement a family of *checked-pointer* classes analogous to the copied and reference-counted object pointers presented in this chapter. They should throw an exception if an attempt is made to use a null pointer. Implement the checked pointers so that they can be used in conjunction with the unchecked versions. How does checking affect efficiency?

14.8 The reference-counted object pointers described in Section 14.5 store the reference count separately from the aggregated object. Design and implement different reference-counted object pointers that store their reference counts in the aggregated object. Discuss the advantages and disadvantages of both approaches in terms of both convenience and performance.

14.9 `CloneableObjPtr<T>` is derived privately from `CopiedObjPtr<T>`, but `CopiedObjPtr<T>` is derived publicly from `CopiedBuiltInPtr<T>`. Does this design make sense? Explain your reasoning.

14.10 Read Notes and Comments 14.6 and the papers it refers to. Write code to determine whether using smart pointers with temporaries works with your compiler.

CHAPTER 15

Classes for Code Organization

THE EXAMPLES of array classes in Chapter 13 and of pointer classes in Chapter 14 demonstrate how to tailor fundamental programming tools in C++. With this chapter we move toward examples that are more specific to scientific and engineering programming.

In this chapter we compare the organization of C++ code to that of a well-written subroutine library, using LAPACK (Linear Algebra PACKage) [5] as the example. LAPACK, the successor to both LINPACK [40] for linear algebra and EISPACK [99] for eigenvector analysis, is written in FORTRAN, runs on most computers, and is available in source code form at no cost; see [5, Section 1.7]. LAPACK's design provides us with a guide for the design of matrix classes, and a high-quality, tested implementation. Consider the code in this chapter as a sketch like one might make early in the design of a program. It illustrates some of the features we want in a family of matrix classes, and we will build on it in later chapters.

15.1 The Organization of LAPACK

Among the many excellent FORTRAN libraries for numerical processing, LAPACK stands out as a well-designed package in wide use and applicable to a variety of scientific and engineering problems. LAPACK consists of some 600,000 lines of commented FORTRAN code, including routines for linear equations, linear least squares problems, eigenvalue and singular value problems, and generalized eigenvalue problems. We shall focus in this chapter on the routines for linear equations, but the ideas are broadly applicable.

LAPACK provides over 250 routines for the solution of the linear equation

$$A\mathbf{x} = \mathbf{b},$$

where A is the coefficient matrix, $\mathbf{b}$ is the right-hand side, and $\mathbf{x}$ is the sought-for solution. Since it is common to need a solution for many right-hand sides, LAPACK actually provides routines to solve the equation

$$AX = B,$$

where the columns of B are the individual right-hand sides and the columns of X are the corresponding solutions.

One way to solve a linear equation is first to factor the coefficient matrix into a product of triangular matrices (and possibly a diagonal or permutation matrix) and then to obtain the solution from the factored form by backward or forward substitution. For example, LAPACK factors a general matrix A into the product of a permutation matrix P, a unit lower triangular matrix L (ones on the diagonal and zeros above the diagonal) and an upper triangular matrix U (zeros below the diagonal). The solutions X are obtained by solving two successive triangular systems of equations:

$$LY = P^{-1}B \qquad \text{and} \qquad UX = Y.$$

The myriad subroutines in LAPACK correspond to various special structures in the matrix A, various factorizations, and various optional computations.

The LAPACK users' guide [5, Section 2.1.3] reveals much of the library's structure in terms of the LAPACK subroutine naming convention. Table 15.1 shows the linear equations portion. The first letter of each subroutine name specifies the type of the matrix elements, the last three letters specify the computation that is to be performed, and the second two letters specify the following aspects of the matrix:

Structure. Matrix structure refers to the appearance of the patterns of zero and nonzero elements illustrated in Fig. 15.1.

Properties. Many matrices that arise in practice have properties that facilitate solving certain problems. A linear equation with a symmetric coefficient matrix can be solved in fewer operations if the matrix is known to be *positive definite* (a matrix A is positive definite if $A = A^T$ and $\mathbf{x}^T A\mathbf{x} > 0$, for all nonzero $\mathbf{x}$). A symmetric matrix not known to be positive definite is called *indefinite*.

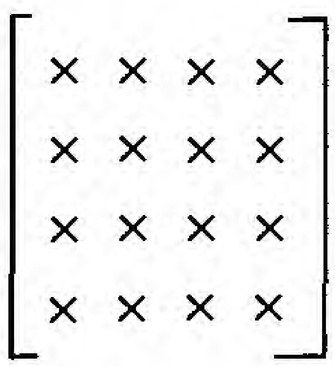
Representation. LAPACK uses four matrix storage schemes. The conventional scheme stores a matrix in a two-dimensional FORTRAN array. The packed scheme stores symmetric, Hermitian, and triangular matrices packed by columns into a one-dimensional FORTRAN array. An $m \times n$ banded matrix

X	*Element Type*
S	REAL
D	DOUBLE PRECISION
C	COMPLEX
Z	COMPLEX*16

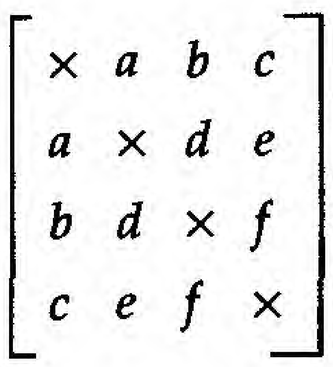
YY	*Matrix Structure*
GB	General band
GE	General
GT	General tridiagonal
PO	Symmetric (Hermitian) positive definite
PP	Symmetric (Hermitian) positive definite packed
PB	Symmetric (Hermitian) positive definite banded
PT	Symmetric (Hermitian) positive definite tridiagonal
SY	Symmetric (complex) indefinite
SP	Symmetric (complex) indefinite packed
HE	Hermitian indefinite
HP	Hermitian indefinite packed
TR	Triangular
TP	Triangular packed
TB	Triangular banded

ZZZ	*Computation*
TRF	Factor
TRS	Solve from factorization
CON	Estimate condition
RFS	Refine solution
TRI	Inverse from factorization
EQU	Equilibrate (scale)

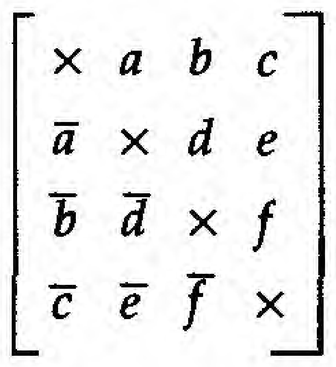
Table **15.1** LAPACK Subroutine Naming Convention. All subroutine names consist of six letters of the form XYYZZZ, where X indicates the type of the matrix elements, YY indicates the structure of the matrix, and ZZZ indicates the computation to be done by the subroutine.



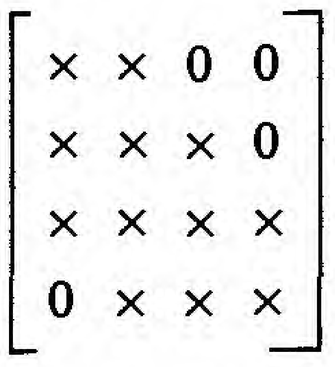
General



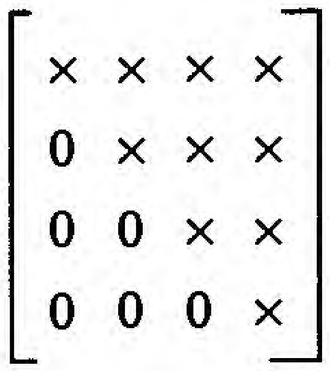
Symmetric



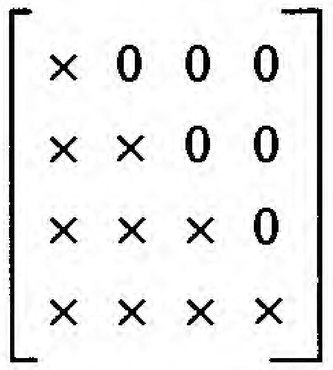
Hermitian



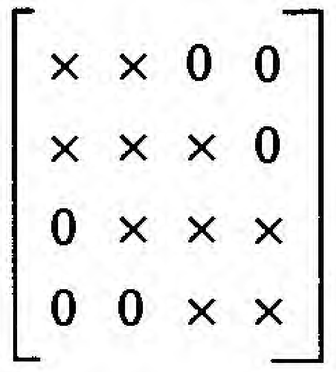
Banded



Upper Triangular



Lower Triangular



Tridiagonal

Figure 15.1 Some LAPACK Matrix Structures. Elements that are not necessarily zero are indicated by × or by a letter. Elements with the same letter have the same value, and an overbar indicates complex conjugation.

is stored in a two-dimensional FORTRAN array smaller than $m \times n$. An (unsymmetric) tridiagonal matrix is stored in three one-dimensional FORTRAN arrays.

Each LAPACK subroutine can be thought of as a point in a carefully organized five-dimensional design space, which has the subroutine's coordinates encoded within its name. The dimensions of the design space—element type, operation, matrix structure, matrix properties, and matrix representation—are almost independent.

The name of each of the subroutines represented in Table 15.2 conveys the nature of both the function the routine performs and the arguments it operates on. Unfortunately a naming convention is not sufficient to allow the compiler to detect programming errors such as calling `SSYTRF` instead of `SSPTRF` to factor a symmetric, indefinite, packed matrix. The remainder of this chapter shows how (aspects of) LAPACK's organization can be expressed in C++, providing both a more concise programming notation and improved error detection.

LAPACK is exceedingly well organized but nevertheless complicated. Therefore we shall explore it in stages, beginning by grouping the data and functions into classes, reformulating the classes to capture the LAPACK optimized storage method, and ending with a different parameterization to express commonality across the library independent of storage method.

	TRF	TRS	CON	RFS	TRI	EQU
SGB	✓	✓	✓			✓
SGE	✓	✓	✓	✓	✓	✓
SGT	✓	✓	✓	✓		
SPO	✓	✓	✓	✓	✓	✓
SPP	✓	✓	✓	✓	✓	✓
SPB	✓	✓	✓	✓		✓
SPT	✓	✓	✓	✓		
SSY	✓	✓	✓	✓	✓	
SSP	✓	✓	✓	✓	✓	
CHE	✓	✓	✓	✓	✓	
CHP	✓	✓	✓	✓	✓	
STR		✓	✓	✓	✓	
STP		✓	✓	✓	✓	
STB		✓	✓	✓		

Table 15.2 Available LAPACK Subroutines. The naming convention is shown in Table 15.1. The initial S in the names may be replaced by D, C, or Z, and the initial C may be replaced by a Z.

15.2 Grouping Data and Functions

Most subroutine libraries contain related subroutines with subsets of identical arguments. In well-designed libraries, such arguments are grouped consistently across all subroutines. In this section, we use LAPACK to illustrate classes as a means of expressing such grouping of arguments.

15.2.1 Factoring and Solving Linear Equations with LAPACK

Consider the paired factoring and solving LAPACK subroutines shown in Table 15.3. Unpacked matrices are passed to the factoring subroutines as a triple or quadruple: The matrix in a FORTRAN array `A`, `A`'s leading dimension `LDA` (see Notes and Comments 15.1), the number of columns `N`, and, for general matrices (which can be rectangular), the number of rows `M`. Upon return, `A` contains the factored result and `IPIV` contains the pivots (where used). The matrix and pivots go back into the solving subroutines along with a matrix of the right-hand sides, passed as the triple `B`, `LDB`, and `N`. The solution to the linear system is returned in `B`. Similar patterns, different in detail, appear in the calling sequences for other LAPACK subroutines.

Options	*Problem Size*	*Matrix*	*Pivots*	*Work Space*	*Right Side*	*Status*
SGETRF(	M,N,	A, LDA,	IPIV,			INFO)
SGETRS(TRANS,	N,NRHS,	A, LDA,	IPIV,		B, LDB,	INFO)
SPOTRF(UPLO,	N,	A, LDA,				INFO)
SPOTRS(UPLO,	N,NRHS,	A, LDA,			B,LDB,	INFO)
SPPTRF(UPLO,	N,	AP,				INFO)
SPPTRS(UPLO,	N,NRHS,	AP,			B,LDB,	INFO)
SSYTRF(UPLO,	N,	A, LDA,	IPIV,	WORK,LWORK,		INFO)
SSYTRS(UPLO,	N,NRHS,	A, LDA,	IPIV,	WORK,LWORK,	B,LDB,	INFO)
CHETRF(UPLO,	N,	A, LDA,	IPIV,	WORK,LWORK,		INFO)
CHETRS(UPLO,	N,NRHS,	A, LDA,	IPIV,	WORK,LWORK,	B,LDB,	INFO)
SSPTRF(UPLO,	N,	AP,	IPIV,			INFO)
SSPTRS(UPLO,	N,NRHS,	AP,	IPIV,		B,LDB,	INFO)
CHPTRF(UPLO,	N,	AP,	IPIV,			INFO)
CHPTRS(UPLO,	N,NRHS,	AP,	IPIV,		B,LDB,	INFO)

Table 15.3 Calling Sequences for Pairs of LAPACK FORTRAN Subroutines. One routine of each pair factors the matrix A (TRF-ending subroutines) and the other routine solves the resulting factored linear systems (TRS-ending subroutines). Subroutines for banded and tridiagonal matrices are omitted.

The form of the factored matrix depends on the kind of matrix. For example, SGETRF uses single precision to factor a general matrix into the product of a permutation (represented by pivots), a unit lower triangular matrix, and an upper triangular matrix. SPSTRF factors positive definite symmetric matrices into the Cholesky factorization [50, Section 4.2] $U^T U$, where U is an upper triangular. There are similar variations for the other kinds of matrices.

Each factoring subroutine overwrites its input matrix with a factored form, but these factored forms are not interchangeable. Each must be passed to its corresponding solving subroutine: If SGETRF factorization is used, then SGETRS must be called for the solution, not SSPTRS. In small programs that are never modified, chances are good that the correct pairs will be used. In larger, more complex programs lasting longer, the chances are good that some incorrect pairs will be used.

The inconvenient repetition of logically coupled matrix arguments and the potential misuse of logically coupled function calls are common problems that classes can solve. By bringing the arguments together and combining them with the functions in an appropriate way, we can create a system of classes for factoring matrices.

15.2.2 Grouping into Classes

Now imagine grouping the common data in the calling sequences of Table 15.3 and being able to give each group a name. We could group the matrix storage and dimensions into a class. Then in each of the subroutines, the list of common arguments for each matrix could be replaced with a single argument specified by the name of the class. The properties of the matrices that are encoded implicitly in the LAPACK subroutine names would become explicit in the class name. For example, the matrices valid as arguments for xGEzzz subroutines—conventionally stored, rectangular matrices—could become LapackRect<T>, whereas those valid for xPPzzz subroutines—packed, symmetric, positive definite matrices—could become LapackSymPosDefPacked<T>, and so on, with T being the matrix element type.

LapackRect<T> objects would contain the matrix storage and the dimensions. Given ConcreteFortranArray2d<T> from Chapter 13, LapackRect<T> might look roughly like this:

ch15/tmpl0.C

```
template<class T>
class LapackRect :
    public ConcreteFortranArray2d<T> { // LAPACK A, M, and N
public:
    LapackRect(Subscript nrows, Subscript ncols);
    FactoredLapackRect<T> factor();
    // ...
};
```

Similarly, given a ConcreteFortranSymmetricPackedArray2d<T> class template (see page 560) that provides storage for a symmetric matrix in packed form, LapackSymPosDefPacked<T> could be defined roughly like this:

ch15/tmpl0.C

```
template<class T>
class LapackSymPosDefPacked :
    public ConcreteFortranSymmetricPackedArray2d<T> {
public:
    LapackSymPosDefPacked(Subscript size);
    FactoredLapackSymPosDefPacked<T> factor();
};
```

Corresponding class templates FactoredLapackRect<T>, FactoredLapackSymPosDefPacked<T>, and so on could be defined for the factored matrices. For example, the factored form of a conventionally stored rectangular matrix might be represented by the following class:

ch15/tmpl0.C

```
template<class T>
class FactoredLapackRect {
public:
    LapackRect<T>& solve(LapackRect<T>&);
private:
    friend FactoredLapackRect<T> LapackRect<T>::factor();
    FactoredLapackRect(Subscript nrows, Subscript ncols);

    ConcreteFortranArray2d<T>   fmat;      // LAPACK "A"
    ConcreteFortranArray1d<int> pivots;    // LAPACK "IPIV"
};
```

Since FactoredLapackRect<T> does not behave like an ordinary matrix, it is not derived from ConcreteFortranArray2d<T>. The only sensible way to create an instance of this class is to factor a LapackRect<T>. Therefore we make the constructor private and give LapackRect<T>'s factor() access to it via the friend declaration. Thus LapackRect<T> controls when FactoredLapackRect<T> objects are created.

Next we move the primary matrix argument out of each subroutine call and in front of the subroutine name and return the result as an object of an appropriate class. For the LAPACK subprograms in Table 15.3, this would give something like the stylized functions shown in Table 15.4. With this scheme, the factor() member function would be called with an instance of an unfactored object and the solve() member function would be called with the factored object in order to solve the linear equations:

ch15/tmpl0.C

```
LapackRect<float> a1(10,10);
// ... fill a1 ...
FactoredLapackRect<float> factored_a1 = a1.factor();
LapackRect<float> b1(10, 1);
// ... fill b1 with right hand side ...
factored_a1.solve(b1);    // Solution is in b1

LapackSymPosDefPacked<float> a2(10);
// ... fill a2 ...
FactoredLapackSymPosDefPacked<float> factored_a2 = a2.factor();
LapackRect<float> b2(10, 1);
// ... fill b2 with right hand side ...
factored_a2.solve(b2);    // Solution is in b2
```

The Lapack matrix classes and their factored siblings contain all of the information held by the arguments to the LAPACK subroutines. We omit LDA because its purpose is to allow a matrix to be stored in a portion of a larger matrix. This stan-

Return Type	*Class*	*Function*
FactoredLapackRect<T>	LapackRect<T>::	factor();
LapackRect<T>&	FactoredLapackRect<T>::	solve(LapackRect<T>
FactoredLapackSymPosDef<T>	LapackSymPosDef<T>::	factor();
LapackRect<T>&	FactoredLapackSymPosDef<T>::	solve(LapackRect<T>
FactoredLapackSymPosDefPacked<T>	LapackSymPosDefPacked<T>::	factor();
LapackRect<T>&	FactoredLapackSymPosDefPacked<T>::	solve(LapackRect<T>
FactoredLapackSymmetric<T>	LapackSymmetric<T>::	factor();
LapackRect<T>&	FactoredLapackSymmetric<T>::	solve(LapackRect<T>
FactoredLapackHermitian<T>	LapackHermitian<T>::	factor();
LapackRect<T>&	FactoredLapackHermitian<T>::	solve(LapackRect<T>
FactoredLapackSymmetricPacked<T>	LapackSymmetricPacked<T>::	factor();
LapackRect<T>&	FactoredLapackSymmetricPacked<T>::	solve(LapackRect<T>
FactoredLapackHermitianPacked<T>	LapackHermitianPacked<T>::	factor();
LapackRect<T>&	FactoredLapackHermitianPacked<T>::	solve(LapackRect<T>

Table 15.4 Stylized Function Declarations for LAPACK-Based Classes. The corresponding LAPACK functions are shown in Table 15.3.

dard FORTRAN programming technique is obviated by C++'s dynamic memory allocation. Thus in their most trivial sense, C++ classes allow related data to be grouped and treated as a single entity, complete with functions.

The design and code that we have presented thus far is inadequate in two major respects:

1. The LAPACK factoring routines overwrite their input with a representation of the factored matrix. Our present design would require two copies of every matrix, which is unacceptable for large matrices. This problem is addressed in Section 15.3.
2. LAPACK matrices come in pairs, unfactored and factored. We have expressed this pairing by a naming convention but have not otherwise captured the common structure in our code. We shall correct this in Section 15.4.

15.3 Indefinite Types with Runtime Failures

From the type point of view, highly optimized subroutine libraries like LAPACK are remarkably intricate. Optimization means taking advantage of special knowledge to avoid work: Optimization requires specialization. Too often, the

special knowledge resides mostly in the programmer's mind or, with luck, in auxiliary program documentation. In this section, we explore techniques for capturing in C++ the sophisticated optimizations used in LAPACK.

LAPACK optimizes space utilization by factoring matrices in place, replacing the contents of the input matrix with the factored form. Since the properties of the factored matrix differ markedly from the original, we would expect that in C++ the original matrix and the factored matrix would have different types. Then we have a conflict: We know that objects have only one type, whereas the LAPACK matrix object would seem to require a metamorphosis.

This conflict highlights the problems we encounter when we begin to transfer to the computer our knowledge about the properties of the objects with which we compute. Consider again the LAPACK subroutine that factors a general rectangular matrix, `SGETRF`. On input, the argument `A` is a general rectangular matrix; on output, `A` will contain (a representation of) two triangular matrices L and U, as was explained in Section 15.1. Before calling `SGETRF`, `A` is manipulated as a rectangular matrix; afterward `A` must be manipulated as an LU-decomposed matrix. The FORTRAN compiler can't possibly deduce the proper way to handle the output matrix. Sooner or later, we will write some code that accesses the elements of `A` incorrectly.

Establishing the types `LapackRect<T>` and `FactoredLapackRect<T>` allows the C++ compiler to detect such erroneous uses, reducing the number of errors in a program. Which type corresponds to the LAPACK variable `A`? Both tempt us: First it is a `LapackRect<T>` and then it is a `FactoredLapackRect<T>`. Neither seems correct: Objects can't change type.

Our solution puts the LAPACK level of functionality, represented by `A`, into a type that is neither `LapackRect` like nor `FactoredLapackRect` like, and then creates interfaces to this object from both `LapackRect<T>` and `FactoredLapackRect<T>`. The conflict is resolved by invalidating the use of one interface object while the other interface object is valid. This approach allows the following code:

ch15/tmpl1.C

```
LapackRect<float> a(10,10);
// ... fill a ...
LapackRect<float> b(10, 1);
// ... fill b ...
FactoredLapackRect<float> factored_a = a.factor();
factored_a.solve(b);                                    // b now contains the solution
```

to work to factor `a` and solve the linear equations. However, once `a` is factored, any further manipulation of the symbol `a`, as in the statement

ch15/tmpl1.C

```
a(1, 2) = 3.1;    // Exception thrown
```

causes an exception to be thrown.

The LAPACK `A`, which at first holds the original matrix and later holds part of the factored form, is represented by a single `FortranArray2d<T>` object (an interfaced version of the `ConcreteFortranArray2d<T>` class template developed in Section 13.2). The two aggregate classes, `LapackRect<T>` and `FactoredLapack-Rect<T>`, now take turns using this single array, providing distinct interfaces to this FORTRAN-level representation. To ensure correct memory management, each aggregate controls the internal array during its turn.

The structure is illustrated before factoring in Fig. 15.2. The unfactored matrix is represented by this class:

ch15/tmpl1.C

```
template<class T>
class LapackRect :
    public virtual Array2d<T> {
public:
    LapackRect(Subscript nrows, Subscript ncols);
    FactoredLapackRect<T> factor();
    // ...

    // Array interface declarations ...
private:
    CopiedObjPtr< FortranArray2d<T> > ap;    // Lapack A
    Boolean                           valid;

    // Check validity -- throw exception if not valid
    void checkValidity() const;
};
```

The class is in the `Array2d<T>` interface category, so its instances behave like two-dimensional arrays. The class has a member that points to a `FortranArray2d<T>` (referential aggregation) so that we can pass control of the array to the factored form by passing the pointer. We use a `CopiedObjPtr<T>` to ensure controlling aggregation (Chapter 14).

When a `LapackRect<T>` object is created, the constructor creates an appropriately sized `FortranArray2d<T>`, initializes `ap` to point to it, and initializes the `valid` flag to true:

ch15/tmpl1.C

```
template<class T>
LapackRect<T>::LapackRect(Subscript nrows, Subscript ncols) :
    valid(Boolean::true), ap(new FortranArray2d<T>(nrows, ncols)) {
}
```

When `factor()` is called, the `valid` flag is set to false and control of the aggregated `FortranArray2d<T>` is handed off to a newly created `FactoredLapackRect<T>`:

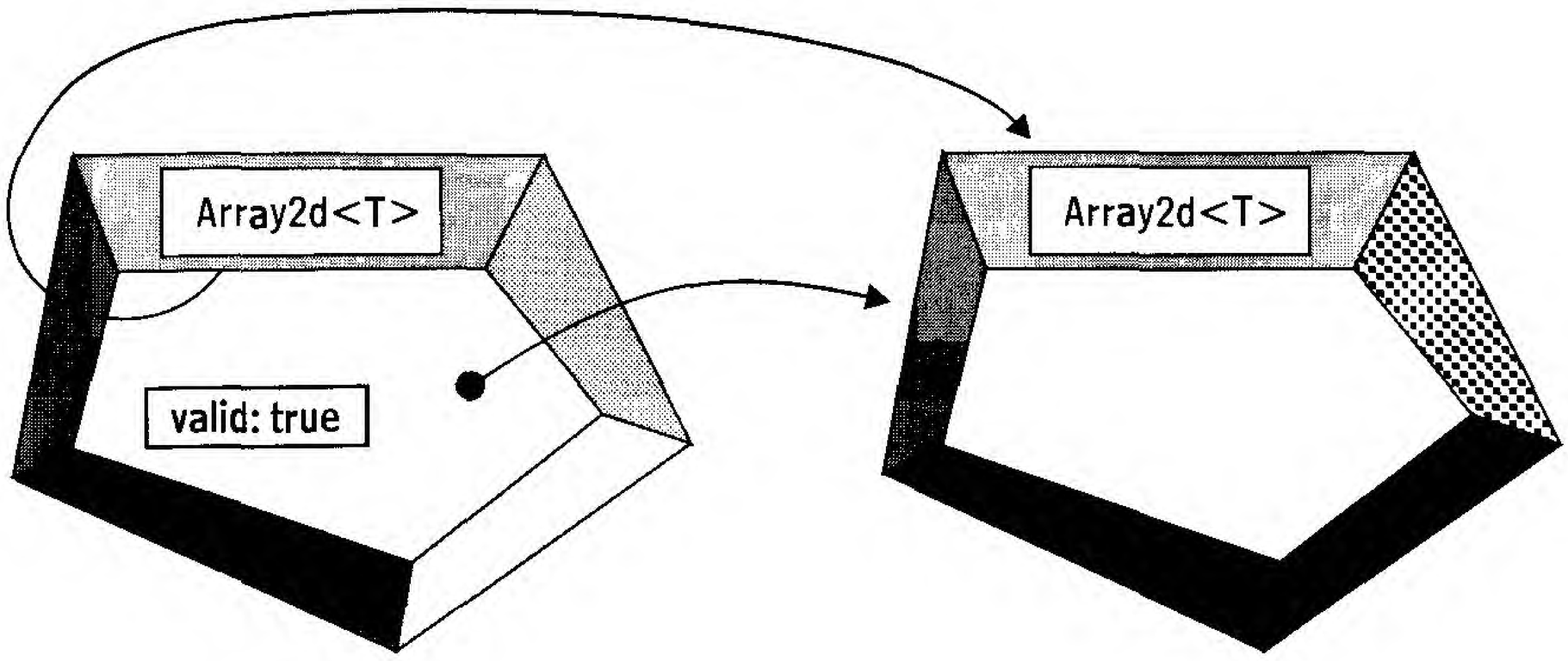


Figure 15.2 The LAPACK Representation Before Factoring. The FORTRAN array that is to be passed to LAPACK is represented by an instance of the class FortranArray2d<T>. Before factoring, the instance is a controlled component of a LapackRect<T>. The array interface of LapackRect<T> is implemented by member function forwarding to the Array2d<T> interface of FortranArray2d<T>.

ch15/tmpl1.C

```
template<class T>
FactoredLapackRect<T> LapackRect<T>::factor() {
    checkValidity();
    valid = Boolean::false;
    return FactoredLapackRect<T>( ap.releaseControl() );
}
```

The releaseControl() member of CopiedObjPtr<T> causes the pointer object to relinquish control of the object it points to, meaning that it will no longer copy or delete that object; it returns a (built-in) pointer to the object. As we will see later, the FactoredLapackRect<T> constructor calls the LAPACK factoring subroutine.

The structure after factoring is shown in Fig. 15.3. The corresponding factored matrix is represented by the following class:

ch15/tmpl1.C

```
template<class T>
class FactoredLapackRect {
public:
    LapackRect<T>& solve(LapackRect<T>&);
private:
    friend FactoredLapackRect<T> LapackRect<T>::factor();
    FactoredLapackRect(FortranArray2d<T>* ap);
```

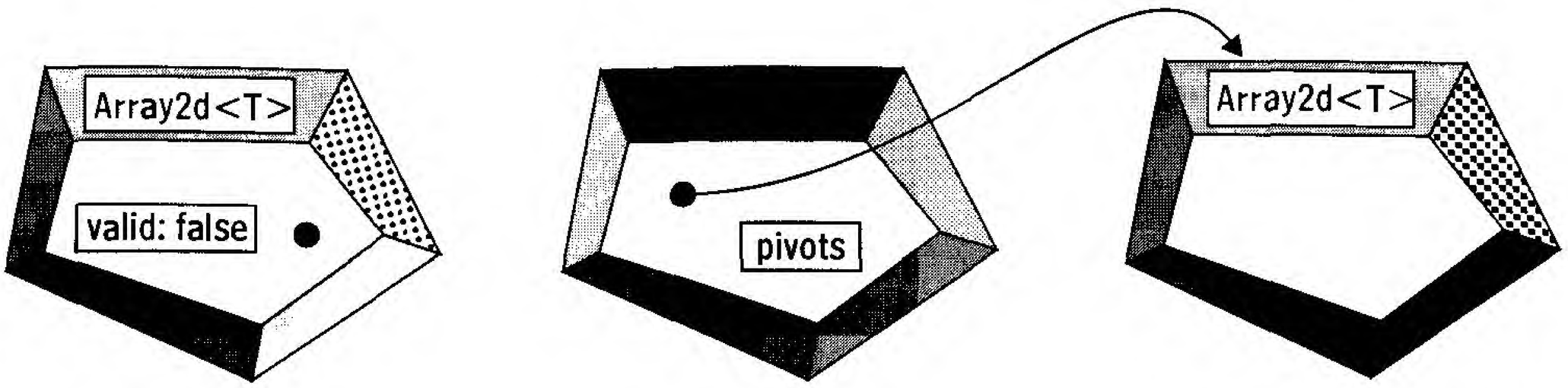


Figure 15.3 The LAPACK Representation After Factoring. The FortranArray2d<T> instance from Fig. 15.2 becomes a controlled component of a FactoredLapackRect<T> and the LapackRect<T> is marked invalid.

```
    CopiedObjPtr< FortranArray2d<T> > fmat_p;      // LAPACK "A"
    FortranArray1d<int>               pivots;      // LAPACK "IPIV"
};
```

The constructor is private, allowing only LapackRect<T>'s factor() member to create a factored matrix. The constructor initializes fmat_p to control the FortranArray2d<T> passed to it, initializes pivots, which is an array of integers used by LAPACK to store the permutation portion of the factored representation, and calls LAPACK to factor the matrix:

ch15/tmpl1.C

```
template<class T>
FactoredLapackRect<T>::FactoredLapackRect(FortranArray2d<T>* ap) :
    fmat_p(ap),
    pivots( min(ap->shape(0), ap->shape(1)) ) {
    // Call LAPACK factoring routine ...
}
```

This implementation allows in-place factorization of a matrix while retaining full type checking. The complexity of the solution reflects the complexity of the information being conveyed to the compiler. Class FortranArray2d<T> represents the level of sophistication in FORTRAN. Accessing a FortranArray2d<T> object requires knowledge of the state of the matrix, which must be verified by the programmer. Programmer errors lead to erroneous results. As we add information—by building typed interfaces like LapackRect<T> and FactoredLapackRect<T>—accessing the matrix requires less information. Whenever we access a matrix, we and the compiler know its structure and the code can be written and compiled accordingly.

Programmer errors then lead to compile-time type-checking errors or runtime error messages but not erroneous results.

This implementation of LapackRect<T> differs from the other classes we have described by moving the type-dependent error testing to runtime. If we wrote an unoptimized version of factor() that copied the contents of the square matrix before factoring it, then the LapackRect<T> object a would always be valid at runtime: We could check for errors at compile time. Once we decided that we wanted the same state—represented by the FortranArray2d<T>—to represent two fundamentally different concepts at runtime, we were forced to cope at runtime with the possibility that the state could be accessed incorrectly.

15.4 Expressing Common Structure

The code we developed in Section 15.3 expresses LAPACK's tight coupling between factored and unfactored matrices using the specific example of LU factorization of rectangular matrices. A glance at Table 15.3 shows that this same pattern is repeated: For every LAPACK matrix type there is a corresponding factored form. To capture this commonality, we need class templates that join all of the unfactored matrices into one category and all of the factored matrices into another category. Then we can pair the categories to express the commonality of structure.

To this end we create a template LapackUnfactored<Rep> for the category of unfactored LAPACK matrices and LapackFactored<Rep> for the factored LAPACK matrices. These templates provide the memory management developed in the preceding section but with the type of the object managed abstracted. The actual representations for specific matrices are given by the class parameter Rep. This class will provide the glue that connects a specific unfactored matrix form to a specific factored matrix form. Thus LapackUnfactored<Rep> will resemble LapackRect<T> if Rep has FortranArray2d<T> for its unfactored form and LapackFactored<Rep> will resemble FactoredLapackRect<T> if Rep adds FortranArray1d<int> to its factored form to hold the LU pivots.

Unfactored matrices are represented by

LapackWrap/Lapack.h

```
template<class Rep> class LapackFactored;    // Declare corresponding factored category

template<class Rep>
class LapackUnfactored :
    public virtual Array2d<Rep::Unfactored::EltT> {
public:
    typedef Rep Representation;              // Factoring representation
    typedef Rep::Unfactored Unfactored;      // Factoring representation's unfactored form
    typedef Rep::Unfactored::EltT EltT;      // Element type
```

```
    LapackUnfactored(Subscript nrows, Subscript ncols);
    LapackFactored<Rep> factor();
    // ...

    // Array interface declarations ...
protected:
    void checkValidity() const;                    // Throw exception if not valid
private:
    CopiedObjPtr<Rep::Unfactored> ap;
    Boolean                       valid;
};
```

Notice the similarity to the definition of the LapackRect<T> class on page 461.

The LapackUnfactored<Rep> template relies on name commonality among the representation classes. The parameter Rep must provide a nested class or type named Unfactored, written Rep::Unfactored, that represents the LAPACK unfactored matrix. Rep::Unfactored must, in turn, provide a nested class or type named EltT that is the type of the matrix elements. Rep must also provide a nested class or type named Factored that represents the corresponding factored form.

The factored form template also relies on name commonality among the representation classes:

LapackWrap/Lapack.h

```
template<class Rep>
class LapackFactored {
public:
    typedef Rep::Unfactored::EltT EltT;                    // Element type

    typedef Rep::Unknowns1d Unknowns1d;                    // Single right-hand-side
    typedef Rep::Knowns1d Knowns1d;

    typedef Rep::Unknowns2d Unknowns2d;                    // Multiple right-hand-sides
    typedef Rep::Knowns2d Knowns2d;

    Rep::Unknowns1d& solve(Rep::Unknowns1d& b);            // Overwrites b with x.
    Rep::Unknowns1d& solve(Rep::Unknowns1d& x, const Rep::Knowns1d& b);

    Rep::Unknowns2d& solve(Rep::Unknowns2d& b);            // Overwrites b with x.
    Rep::Unknowns2d& solve(Rep::Unknowns2d& x, const Rep::Knowns2d& b);

    Rep::Factored& rep() { return *ap; }
    const Rep::Factored& rep() const { return *ap; }
```

```
protected:
    friend LapackFactored<Rep> LapackUnfactored<Rep>::factor();
    LapackFactored(Rep::Factored* p) : ap(p) {}
private:
    CopiedObjPtr<Rep::Factored> ap;
};
```

Again notice the similarity to the definition of FactoredLapackRect<T> on page 462. One difference is that we now have four solve() functions, two solve for a single right-hand side and two for multiple right-hand sides (cf. Section 15.1). For each case, one solve() overwrites the right-hand side(s), and the other puts the solution in the argument x. The LapackFactored<Rep> adds Unknowns1d, Unknowns2d, Knowns1d, and Knowns2d to the list of common names required in Rep.

Here is the Rep class for rectangular matrices, using LU decomposition for the factoring algorithm:

LapackWrap/RectLURep.h

```
template<class T>
class RectLURep {
public:
    typedef ConcreteFortranArray2d<T> Unfactored;
    typedef ConcreteFortranArray1d<T> Unknowns1d;
    typedef ConcreteFortranArray1d<T> Knowns1d;
    typedef ConcreteFortranArray2d<T> Unknowns2d;
    typedef ConcreteFortranArray2d<T> Knowns2d;

    class Factored {
    public:
        Factored(RectLURep<T>::Unfactored* mp);
        void solve(ConcreteFortranArray2d<T>&);
        void solve(ConcreteFortranArray2d<T>& x, const ConcreteFortranArray2d<T>& b);
        void solve(ConcreteFortranArray1d<T>&);
        void solve(ConcreteFortranArray1d<T>& x, const ConcreteFortranArray1d<T>& b);
    private:
        CopiedObjPtr< RectLURep<T>::Unfactored > facmat_p;    // Factored matrix
        ConcreteFortranArray1d<int>              pivots;      // Pivots
    };
};
```

RectLURep<T>::Unfactored is defined using a typedef to provide the required name for FortranArray2d<T>. The representation of the factored form, defined by the nested class RectLURep::Factored, aggregates the FORTRAN-level matrix and an array of integer pivots.

A representation for symmetric positive definite packed matrices is similar, but no pivot array is needed:

LapackWrap/SymPosDefPackedLURep.h

```
template<class T>
class SymPosDefPackedLURep {
public:
    // These types are used for Ax=b in this implementation:
    typedef ConcreteFortranSymmetricPackedArray2d<T> Unfactored;      // A
    typedef ConcreteFortranArray1d<T> Unknowns1d;                     // x
    typedef ConcreteFortranArray1d<T> Knowns1d;                       // b
    typedef ConcreteFortranArray2d<T> Unknowns2d;                     // x
    typedef ConcreteFortranArray2d<T> Knowns2d;                       // b

    class Factored {
    public:
        Factored(SymPosDefPackedLURep<T>::Unfactored* mp);
        void solve(ConcreteFortranArray1d<T>&);
        void solve(ConcreteFortranArray1d<T>& x, const ConcreteFortranArray1d<T>& b);
        void solve(ConcreteFortranArray2d<T>&);
        void solve(ConcreteFortranArray2d<T>& x, const ConcreteFortranArray2d<T>& b);
    private:
        // Factored matrix pointer
        CopiedObjPtr< SymPosDefPackedLURep<T>::Unfactored > facmat_p;
    };
};
```

In Section 18.7 we use the same technique for the singular value decomposition factorization using rectangular arrays.

Using class templates to express the pairing of unfactored and factored matrices organizes and regularizes the implementation of a class library for linear systems. But the impact on the organization of code that uses the class library can be even more significant: We can now write code that is independent of a particular class in a category. As a trivial example, let's write analogs of LAPACK's simple driver routines [5, Section 2.2.1]; these solve $AX = B$ using the appropriate LAPACK subroutines:

ch15/Driver.C

```
template<class LapackMatrix, class EltT>
LapackUnfactored< RectLURep<EltT> >
xForAxEqb(LapackMatrix& matrixA, ConcreteFortranArray2d<EltT>& b) {
    return matrixA.factor().solve(b);
}
```

This template function calls different factor and solve functions depending on the structure of the matrix supplied as the LapackMatrix argument and depending on the element type supplied in the LapackUnfactored< RectLURep<EltT> > class.

If we use it like

ch15/Driver.C

```
LapackUnfactored< RectLURep<double> > a1(2, 2);
a1(0,0) = 2; a1(0,1) = 3;
a1(1,0) = 5; a1(1,1) = 6;

ConcreteFortranArray2d<double> b1(2, 1);
b1(0,0) = 4;
b1(1,0) = 7;

cout << xForAxEqb(a1, b1) << endl;
```

then we will decompose a square matrix (calling LAPACK's DGETRF()) and solve for the unknowns (calling LAPACK's DGETRS()), but if we use it like this

ch15/Driver.C

```
LapackUnfactored< SymPosDefPackedLURep<float> > a2(2, 2);
a2(0,0) = 2; a2(0,1) = -2;
a2(1,1) = 5;

ConcreteFortranArray2d<float> b2(2, 2);
b2(0,0) = -2; b2(0,1) = 2;
b2(1,0) = 8; b2(1,1) = -8;

cout << xForAxEqb(a2, b2) << endl;
```

different code will be executed (LAPACK's SPPTRF() and SPPTRS()). The same source code, xForAxEqb(), applies to all members of the LapackUnfactored<Rep> category. Once again categories compress the amount of source code we need to organize as users at the same time that they reorganize the structure of the code provided in libraries.

15.5 Summary

We have compared a C++-style LAPACK class system and the published FORTRAN version as a means of illustrating the impact of C++ on the organization of our programs. Beginning with the LAPACK subroutine library, we broke it down by data structure to create classes, by element type to create template categories, and by matrix structure and properties to create additional template categories.

It might seem that we have written many declarations just to define a few operations on matrices. We have, but the result is worth the effort. First of all, we have captured in C++ code several kinds of relationships among the various kinds of matrices and operations provided by LAPACK. Since the C++ compiler has all of this information, it can help us in several ways. No naming convention is needed to encode matrix type information in function names. This means that it is no longer necessary to change the names of library subroutines called simply to change the precision of the matrices; this helps us write portable code. We can also be sure that the calling sequences for all of our operations are consistent across all kinds of matrices (i.e., rectangular, symmetric, Hermitian, and triangular) and across all matrix element types because the compiler enforces this.

Using LAPACK to illustrate the issues of class organization freed us from some of the problems faced in designing a system of classes from scratch. The LAPACK authors already defined the scope of the problem and selected the key objects and key interactions. To create classes from their work was relatively easy. What the LAPACK code does not contain are relationships among the classes. These relationships are detailed in the LAPACK user's guide as summarized in Table 15.1; we could move relatively quickly by building on this work. Not all C++ projects will have this advantage, but many will: There is, after all, an enormous amount of existing computer code, the only flaw of which is its lack of generality.

By adapting the structure of existing subroutine libraries, we can move more quickly to a set of class definitions; by calling these subroutines in our function definitions, we can go quickly to a working system. We recommend this approach when it is possible. There are zillions of lines of good FORTRAN and C code in the world: Use them. We shall explore the details of calling such code in Chapter 18, where we will call LAPACK subroutines to implement the member functions declared in this chapter.

15.6 Notes and Comments

15.1 The concept of the leading dimension of a FORTRAN array comes from a programming practice used with the BLAS and elsewhere. FORTRAN-77 does not allow the size of an array to be determined at runtime. To write a program that can work with arrays of various sizes, determined at runtime, you declare an array that is as big as the biggest reasonable array your program can handle. Then when the program runs, it stores its data into as much of the array as is needed. When the array is passed to a subroutine, the subroutine needs to know not only the logical size of the array, but also the first dimension of the array as declared; the first or leading dimension gives the stride (cf. Section 13.2.1) information needed to access elements of the array.

15.2 In designing our `LapackUnfactored<Rep>` and `LapackFactored<Rep>` classes, we chose to expose that LAPACK has two fundamentally different representations for each kind

of matrix: unfactored and factored. We made this decision because factored and unfactored matrices have drastically different behaviors. For example, an element of an unfactored, rectangular matrix can be changed in constant time (i.e., under reasonable assumptions, time independent of the matrix size), whereas changing an element of an LU-factored matrix requires time that is a function of the matrix size. However, for some purposes it might be convenient to hide such matters. See Exercise 15.5.

15.7 Exercises

15.1 Use the classes described in this chapter to write the code to solve the following system of equations [50]:

$$3x_1 + 5x_2 = 9$$
$$6x_1 + 7x_2 = 4$$

15.2 Referring to Table 15.3, the LAPACK variable IPIV is a *pivot* created during the matrix factoring, INFO is a status return code, and WORK is a work space. Which of these quantities belong in objects? Which objects?

15.3 The class FactoredLapackRect<T> on page 462 aggregates the LAPACK A matrix using a CopiedObjPtr<T> but incorporates the LAPACK IPIV array as a member. Why?

15.4 Table 15.5 gives the LAPACK calling sequences for the decomposition $A = Q^H T Q$ of a symmetric matrix A into an orthogonal matrix Q and a tridiagonal matrix T. This is the first step to computing the eigenvectors of A. UPLO is 'U' for upper triangle storage of A and 'L' for lower. On input A is the matrix of size N squared; on output A, combined with TAU, defines Q. The outputs D and E together define T. These outputs can be used in a variety of LAPACK routines. For example, SSTERF(N, D, E, INFO) overwrites D with the eigenvalues of the original matrix. Design a set of classes to represent the functionality of these subroutines.

15.5 Design a system of classes that hides the distinction between unfactored and factored matrices. Would it be advantageous to build on the classes described in this chapter, or would it be better to start anew? Why?

15.6 In Section 15.3 we passed the lifetime control from one object to another using the combination of a Boolean and a CopiedObjPtr<T>. Design, implement, and test a pointer class that expresses this combination as a single object. Eliminate the Boolean storage in favor of setting the pointer to null as a flag.

15.7 In Section 15.4 LapackUnfactored<Rep> acts to interface the array pointed to by ap. Based on the interfaces in Chapter 13, complete the interfacing we omitted in the text.

15.8 Create an array class with some of the properties of the pointer class from Exercise 15.6: Give it a releaseControl() member that returns a pointer to an array and then sets a flag that ensures runtime exceptions rather than erroneous values.

Options	Problem Size	Matrix	Tridiagonal	Scale Factors	Work Space	Status
SSYTRD(UPLO	N,	A, LDA,	D,E,	TAU,	WORK,LWORK,	INFO)
DSYTRD(UPLO	N,	A, LDA,	D,E,	TAU,	WORK,LWORK,	INFO)
CHETRD(UPLO	N,	A, LDA,	D,E,	TAU,	WORK,LWORK,	INFO)
ZHETRD(UPLO	N,	A, LDA,	D,E,	TAU,	WORK,LWORK,	INFO)
SSPTRD(UPLO,	N,	AP,	D,E,	TAU		INFO)
DSPTRD(UPLO,	N,	AP,	D,E,	TAU		INFO)
CHPTRD(UPLO,	N,	AP,	D,E,	TAU		INFO)
ZHPTRD(UPLO,	N,	AP,	D,E,	TAU		INFO)

Table 15.5 Calling Sequences for LAPACK FORTRAN Tridiagonal Decomposition Subroutines. The tridiagonal decomposition of a matrix A is $Q^H A Q = T$ where Q is orthogonal. The first letter of the routine gives the precision as in Table 15.1.

15.9 Repeat Exercise 15.7 using the results of Exercise 15.8 and the InterfacedArray2d<T> class from Chapter 13 rather than hand coding.

CHAPTER 16

Algebraic Structure Categories

To help exploit the power of C++ operators, we introduce in this chapter a system of classes we call the *algebraic structure categories*. They define relationships between C++ operators and other functions using the mathematics of algebraic structures. In addition to being useful for adding operators to classes, the code here gives further examples of operators as functions and provides an extended example of the function-structure category construct we introduced in Chapter 12.

If we view the instances of a class as a set of objects, then some of the operators or functions in the class may combine objects to yield a new object in the set. For example, ordinary addition and multiplication of integers are ways of combining integers. Vector addition and cross product are ways of combining vectors. Combining operations need not be numerical. For example, group theory, the basis for many advances in elementary particle theory and quantum chemistry, uses abstract geometrical operations. Algebraic structures describe the consequences of interactions among combining operations.

Mathematicians have explored operators and their interrelations in detail. We can capture that knowledge for our programs by translating the mathematical ideas into C++. This parallels the idea of translating numerical algorithms into FORTRAN, but it occurs at a more abstract level. As a programming device, the abstract algebra categories organize the arithmetic operators into a consistent system based on well-recognized rules.

The algebraic structure categories define complete and consistent sets of operators. We can express the algebraic structures once in C++ and use them over and over again as we write new classes, reducing the cost of providing a complete set of operators for a new class.

We begin in Section 16.1 with an introduction to abstract algebra, mainly its terminology and basic concepts. Section 16.2 gives a simple example of a set with algebraic structure, complex numbers, to see how our use of abstract algebra will appear in C++ code. The implementation of our abstract algebra categories begins

in Section 16.3 with single-set algebraic structures and continues in Section 16.4 with two-set structures. Section 16.5 applies the algebraic structure categories to dimensonal analysis and Section 16.6 uses them to add arithmetic to the array classes of Chapter 13.

16.1 Algebraic Structures

Let $a, b, c, \ldots$ be elements of a set S. An *internal binary composition law* is a function that assigns to any ordered pair (a, b) in $S \times S$ an element c of S. Using $\square$ to denote a composition law, we write binary composition as $a \square b = c$. Familiar examples of binary composition laws are addition and multiplication on the set of real numbers. The simplest algebraic structures consist of one set and one internal composition law. More general structures are obtained by increasing the number of composition laws and the number of sets. Various properties of the composition laws induce different algebraic structures on the sets.

16.1.1 Structures over One Set

Algebraic structures over one set are characterized by four properties: associativity, commutativity, identity, and inverse. The structures resulting from various combinations of these properties are listed in Table 16.1.

A composition law is *associative* if $a \square (b \square c) = (a \square b) \square c$ for all $a, b, c \in S$. A *semigroup* consists of a nonempty set S and an associative composition law $\square$.

Algebraic Structure	*Multiplicative Law*			*Additive Law*		
	Associative	*Identity*	*Inverse*	*Associative*	*Identity*	*Inverse*
Semigroup	√					
Monoid	√	√				
Group	√	√	√			
Abelian semigroup				√		
Abelian monoid				√	√	
Abelian group				√	√	√
Ring	√			√	√	√
Ring with unit	√	√		√	√	√
Field	√	√	√	√	√	√

Table 16.1 Algebraic Structures over a Single Set.

Positive integers with multiplication form a semigroup, as do nonempty character strings with string concatenation as the composition law.

A *monoid* is a semigroup with a distinguished *unit* or *identity* element, denoted 1, having the property that $1 \square a = a \square 1 = a$. Character strings, including the empty character string, with string concatenation as the composition law form a monoid, as do the nonnegative integers under addition.

A *group* is a monoid with the additional property that every element a has an inverse a^{-1} such that $a \square a^{-1}$ is the identity element 1. The integers under addition form a group, with negation as the inverse and with zero at the additive identity.

A *commutative*, or *Abelian*, algebraic structure has a commutative composition law, a composition law such that $a \square b = b \square a$ for any two elements a and b. The set of integers under addition is an Abelian structure (an Abelian group); the set of 3×3 matrices with integer elements under matrix multiplication is a non-Abelian structure (a non-Abelian monoid). Sometimes the term *additive* is used instead of commutative or Abelian, and the abstract composition law of an Abelian algebraic structure is often written as $+$; similarly, a noncommutative structure is often called *multiplicative* and its composition law is denoted by $*$.

More general structures consist of two or more internal composition laws, analogous to addition and multiplication. A *ring* has two composition laws, an addition law and a multiplication law, related by the distributive laws, $a(b+c) = ab+ac$ and $(a+b)c = ac+bc$. A ring, then, is a combination of an Abelian group and a semigroup, with the addition law taken from the Abelian group and the multiplication law from the semigroup.

If the semigroup is a monoid (has an identity), the ring is called a *ring with unit*. If the multiplication law is commutative, the ring is called a *commutative ring* or *commutative ring with unit*, as appropriate. The set of even integers under addition and multiplication is a commutative ring (without unit), whereas the set of integers under addition and multiplication is a commutative ring with unit. The set of 3×3 matrices with integer elements under matrix addition and matrix multiplication is a (noncommutative) ring with unit.

The final and most powerful algebraic structure with composition laws over one set is the *field*, a commutative ring with unit with the additional property that every nonzero element (element that is not the additive identity) has a multiplicative inverse. In other words, the nonzero elements form a group under the ring's multiplication. Real numbers under addition and multiplication are a field, as are complex numbers under complex addition and multiplication. Other examples of algebraic structures over one set appear in the Exercises at the end of this chapter.

16.1.2 Structures over Two Sets

In scientific and engineering computing, we often work with structures having two sets, sets we think of as vectors and scalars. Let S and V denote two sets,

Algebraic Structure	*Vectors*	*Scalars*
Linear space	Abelian group	Field
Linear algebra	Ring	Field
Algebra with unit	Ring with unit	Field
Division algebra	Field	Field

Table 16.2 Algebraic Structures over a Set of Vectors and a Set of Scalars

each separately having one of the single-set algebraic structures described earlier. Having two sets opens the possibility of an *external composition law*, a composition law that assigns to any ordered pair (v, s) in $V \times S$ an element c of V. Now we have the internal composition laws within each set and an external composition law across the two sets, each with various properties. Once again the various properties induce different structures on the sets. The ones of most interest to us are listed in Table 16.2.

A *linear space*, commonly called a *vector space*, is an algebraic structure that consists of two sets, V and S, and an external composition law. S is a commutative field, called the *scalars*, and V is an Abelian group, called the *vectors*. The n-vectors of real numbers, for fixed n, form a linear space where addition of vectors is element-by-element addition and scalar multiplication is multiplication of each element by the scalar. Likewise, $n \times n$-matrices under element-by-element addition and scalar multiplication by reals is a linear space. Substitute complex numbers for the real number elements in either of these examples and we have another linear space. The set of polynomials under polynomial addition, with scalar multiplication by reals being multiplication of the polynomial coefficients by the scalar, is also a linear space.

Linear spaces are not rich enough to represent structures encountered in scientific programming. For example, matrices have two internal composition laws—matrix addition and matrix multiplication—but the vectors in a linear space have only one internal composition law. A *linear algebra* (or, simply, an *algebra*) is a linear space in which the set of vectors forms a ring; an *algebra with unit* is a linear space in which the vectors form a ring with unit. A *division algebra* is a linear space in which the vectors form a field. Once the linear space has inversion, we can define division of a scalar by a vector.

Given one or two sets and their composition laws, the process of deciding which structure they have is summarized in the decision tree shown in Fig. 16.1. To use the tree, start with the set (for two-set structures, start with the set of vectors) and ask whether its composition law is multiplicative or additive; follow the corresponding branch. Continue asking the questions at each join, stopping when you reach a leaf of the tree.

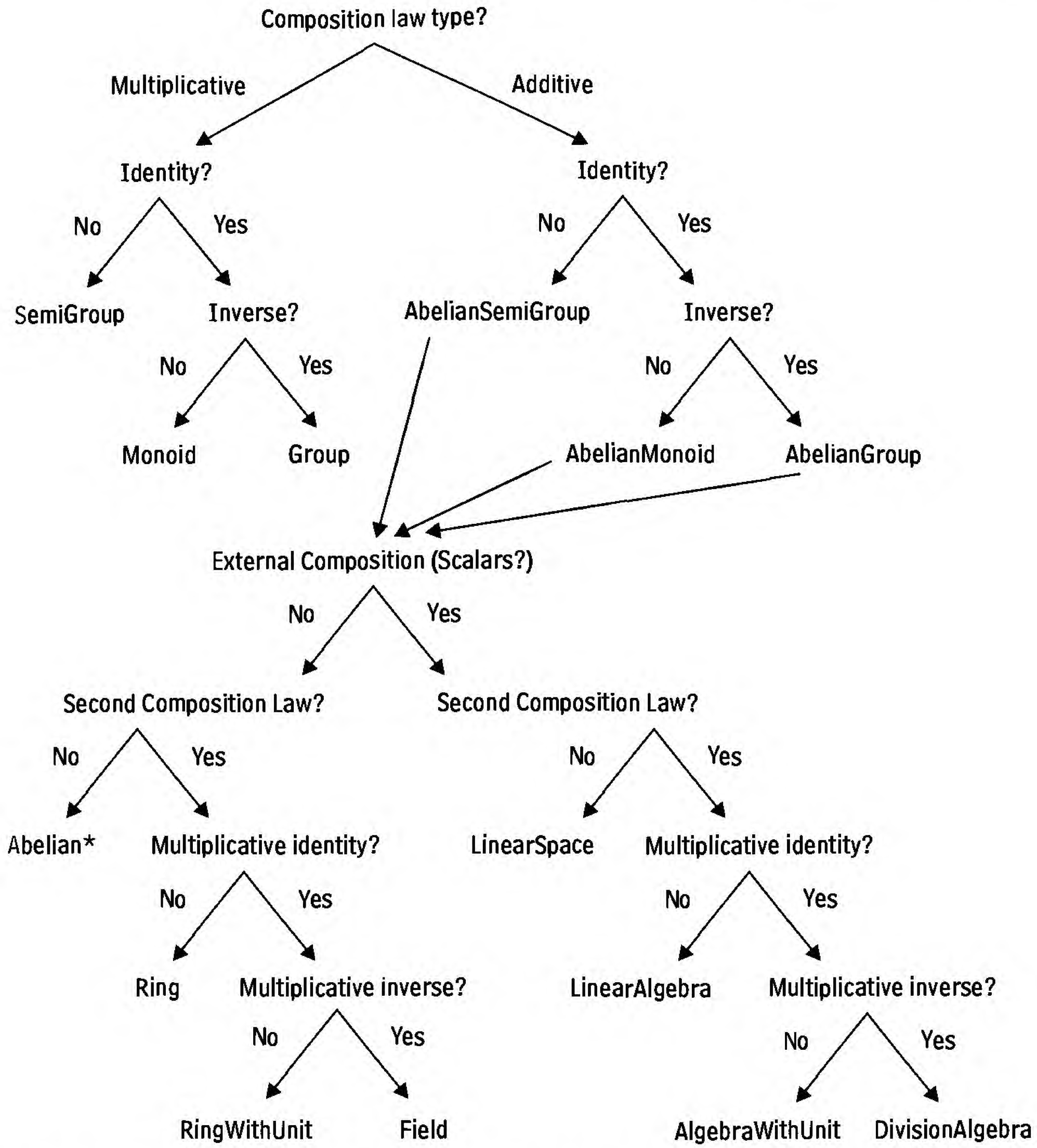


Figure 16.1 Decision Tree for Choosing an Algebraic Structure. The leaves of the decision tree represent categories, with the leaf Abelian* standing for the Abelian structure before the test for external composition laws.

We have done little more in this section than give definitions and a few examples of algebraic structures. The topic is vast and beautiful but well beyond the scope of this book. Nevertheless the structures we have introduced here will be useful throughout the rest of this book—once we figure out how to express them in C++.

16.2 Example: ComplexFloat

What do algebraic structures mean in a C++ program and how can we use them? Suppose that our set consists of all of the instances of some class, say `T`. Then our composition law could be defined by a function that takes two instances of `T` and returns the instance specified by the composition law. This would correspond directly to the mathematical notion of an algebraic structure. For programming we must also know

- How do we give a class an algebraic structure?
- What name or operator do we use for the composition law(s)?
- Are the arguments altered?
- In typical C++ usage, we expect that certain operators (e.g., `*` and `*=`) are related to each other. How do we ensure that the necessary conventions are obeyed?

Algebraic structure classes provide uniform answers to these questions. A user of these classes supplies a function that implements each composition law. The algebraic structure classes supply the rest.

The algebraic structure classes contain function structure common to a category of classes. For example, we will define a `FieldCategory<T>` class that will define functions consistent with the mathematical notion of a field. A class `T` joins in the category of classes that have field algebraic structure by deriving from `FieldCategory<T>`, a technique we called a function-structure category in Section 12.7. The composition law `+=` will be defined by `T`, and the category-defining class `FieldCategory<T>` will supply the `operator+()`, defined to provide the usual relationship between the `+=` and `+` operators: `t1 + t2` is equivalent to `T tmp(t1); tmp += t2;`.

Let's look at a specific example where `T` is `ComplexFloat`. Most C++ compilers come with a `complex` class, usually defined in the header `complex.h`, that implements complex numbers and complex arithmetic with the real and imaginary parts each represented by a `double`. Suppose that we are going to work with large arrays of complex numbers and don't need double precision: We want complex numbers with the real and imaginary parts each represented by a `float`. We want a `ComplexFloat` class.

Referring to Fig. 16.1, we first ask whether complex numbers have an additive composition law; they do—complex addition—so we follow the right branch of the decision tree. Complex addition has both an identity—zero—and an inverse—negation of the real and imaginary parts. Since we are working with one set—complex numbers—there is no external composition. There is a second composition law—complex multiplication—and it has both an identity and an inverse for all nonzero elements. Thus complex numbers form a field.

Now we want to express the field structure of ComplexFloat in C++. Each algebraic structure we use is expressed as a function-structure category, a base class parameterized by the target class itself. To put class ComplexFloat in the field category, we derive ComplexFloat from FieldCategory<ComplexFloat>. ComplexFloat must define the multiplicative and additive composition laws, as members operator*=() and operator+=(), respectively; the multiplicative and additive identities, as members setToOne() and setToZero(), respectively; and the multiplicative and additive inverses, as members operator/=() and operator-=(), respectively. Here is the class definition:

Algebra/ComplexFloat.h

```
class ComplexFloat :
    public FieldCategory< ComplexFloat > {
public:
    ComplexFloat(float r, float i = 0.) : real_part(r), imag_part(i) {}
    ComplexFloat() {}

    // Convert to complex numbers from <complex.h>
    operator complex() const { return complex(real_part, imag_part); }

    float   real()  const { return real_part;}
    float& real()         { return real_part;}
    float   imag() const { return imag_part;}
    float& imag()         { return imag_part;}

    ComplexFloat& conj() { imag_part = -imag_part; return *this; }

    // Field "user must provide"
    inline ComplexFloat& operator*=(const ComplexFloat& rhs);
    ComplexFloat& operator/=(const ComplexFloat& rhs);
    inline ComplexFloat& setToOne();

    inline ComplexFloat& operator+=(const ComplexFloat& rhs);
    inline ComplexFloat& operator-=(const ComplexFloat& rhs);
    inline ComplexFloat& setToZero();
```

```
private:
    float real_part;
    float imag_part;
};
```

The operator*=() member function is defined on page 486 and the rest of the function definitions are left as an exercise (Exercise 16.3).

Given definitions for the six functions declared in the ComplexFloat class, the field category provides the other functions dictated by consistency with C++. This includes the binary operators +, −, *, /, the unary operators −, ++, and −−, and the functions pow(), invert(), repeat(), and negate() discussed in the next section.

Using this class we could write, for example, a class for the linear fractional transformation

$$w(z) = \frac{az + b}{cz + d}.$$

This formula transforms circles into scaled and shifted circles in the extended complex plane. The class could look like this:

ch16/lin-frac.C

```
class LinearFractionalTrans {
public:
    LinearFractionalTrans(ComplexFloat a, ComplexFloat b, ComplexFloat c, ComplexFloat d) :
        _a(a), _b(b), _c(c), _d(d) {}
    ComplexFloat operator()(ComplexFloat z) const {
        return (_a * z + _b) / (_c * z + _d);
    }
private:
    ComplexFloat _a, _b, _c, _d;
};
```

All of the operators used in the definition of operator()() were defined by the Field-Category<Complex Float>.

To understand the algebraic structure categories themselves and to see how they work as a system, we now examine the category classes in detail. More examples of their use then follow.

16.3 Categories for Single-Set, Single-Law Structures

We begin our tour of the algebraic structure categories with the simplest categories, those for a single set with a single composition law. We implement these algebraic structure categories using the function-structure category technique of

Section 12.7. Recall that a function-structure category uses a base class parameterized by the derived-class type. This allows the base class to declare and define functions that take derived-class type arguments and return derived-class type objects.

For example, in the algebra system we declare an operator+() taking two objects from the derived class and returning their sum. These functions can then be inherited by the derived class. The base class functions can call derived-class functions. For example, our algebra system will call operator+=() inside of the definition for operator+(); derived classes must supply the operator+=(). The real work occurs in these derived-class functions. The function-structure base class uses name commonality to establish the rule that c=a+b is the same as c=a; c+=b. Although we shall focus on the use of function-structure categories for operators, function-structure commonality need not involve operators at all.

16.3.1 Description

The single-set categories with a multiplicative composition law are listed in Table 16.3. In the left column are base classes parameterized by the class name for the set of objects with algebraic structure. The categories are arranged in the class DAG shown in Fig. 16.2. The DAG reflects directly the relationships among the algebraic structures themselves: A group has the functions of a monoid plus a multiplicative inverse; a monoid has the functions of a semigroup plus a unit (identity element). The member functions provided by a category are inherited by

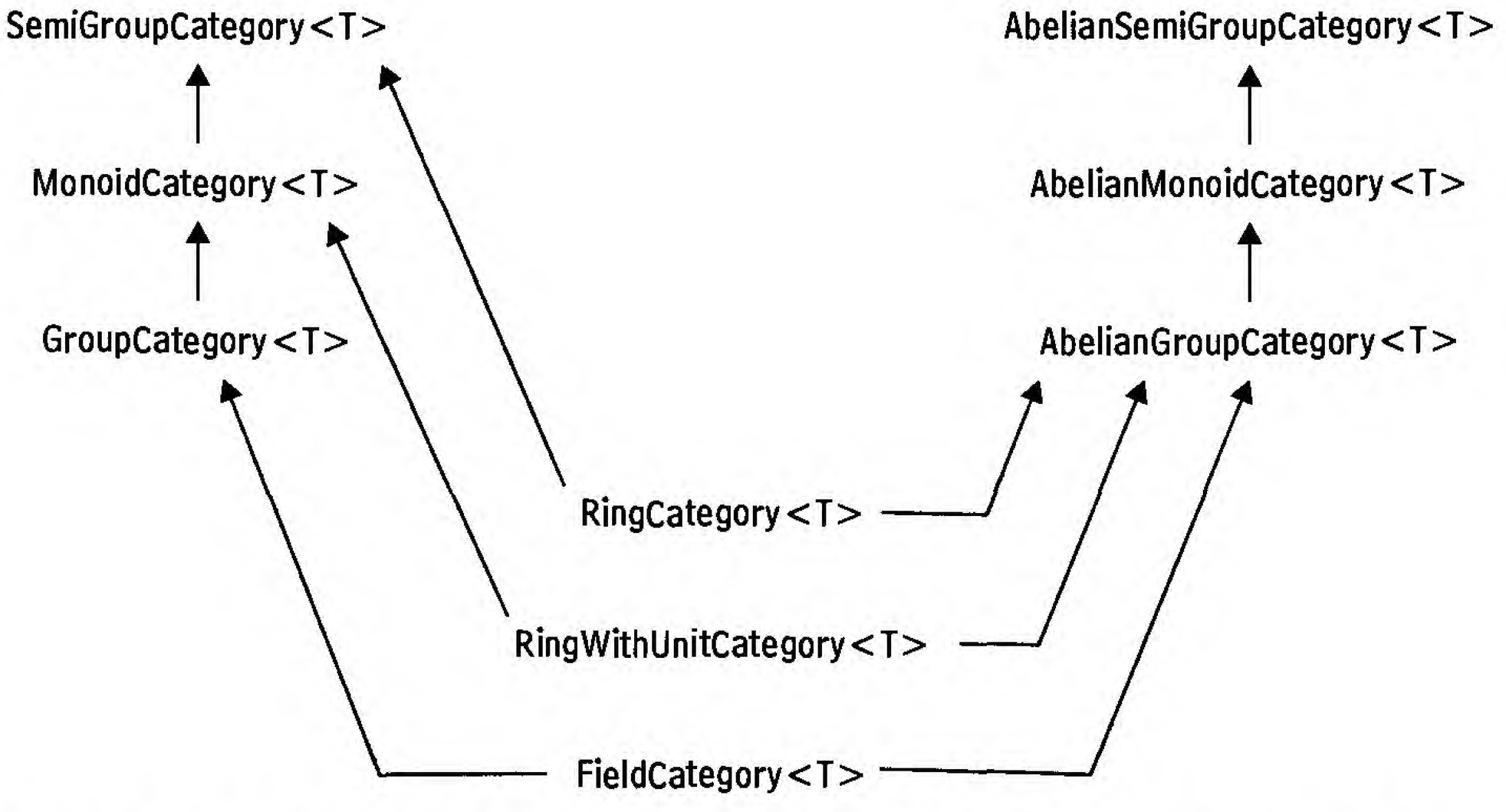


Figure 16.2 Class DAG for One-Set Algebraic Structures. T is the class given the algebraic structure.

Category	*User-Must-Define Functions*	*Category-Provides Functions*
SemiGroupCategory<T>	T& operator*=(const T&)	T& pow(Positive<int>) T ::pow(const T& , int) T ::operator*(const T& , const T&)
MonoidCategory<T>	T& setToOne()	T& pow(NonNegative<int>)
GroupCategory<T>	T& operator/=(const T&)	T& pow(int) T& invert() T ::inverse(const T&) T ::operator/(const T& , const T&)

Table 16.3 Categories for Single-Set Algebraic Structures with a Multiplicative Composition Law. Each category provides the functions listed for it and for the categories listed above it in the table. Functions preceded by the scope operator :: are global; others are member functions.

derived categories and hence by the class T. Thus a class in one of the categories gets all of the functions listed in Table 16.3 for that category and those listed in the categories above it in the table.

The functions listed in the "User-Must-Define Functions" column are the functions that the target class, the class derived from the category, must supply to define the composition law and its properties. These functions are all members that modify the content of the *this object and then return *this.

The functions listed in the "Category-Provides Functions" column are provided by the category and are defined in terms of the user-must-define functions. The pow() functions are shorthand for repeated application of the composition law—repeated multiplication—such that a.pow(3) is the same as a*a*a. For a semigroup, the repetition count must be positive because there is no unit; monoids introduce a unit, allowing a zero repetition count as well; groups introduce an inverse, allowing a negative repetition count too—repeated composition followed by inversion. The classes Positive<int> and NonNegative<int> do what their names imply. Their implementation is described in Section 16.3.2.

The pow() member function in the group category hides the pow() member function in the monoid category, which in turn hides the pow() member function in the semigroup category. However, there can be only one global pow() function, taking an integer count. Overloading the global function on the type of the count would lead to ambiguous conversions: Given an instance of a class in the monoid category, the compiler wouldn't be able to choose between converting a power of 2 to a positive or to a nonnegative integer. Our global pow() function takes

Category	*User-Must-Define Functions*	*Category-Provides Functions*
AbelianSemiGroupCategory<T>	T& operator+=(const T&)	T& repeat(Positive<int>) T ::repeat(const T& , int) T ::operator+(const T& , const T&)
AbelianMonoidCategory<T>	T& setToZero()	T& repeat(NonNegative<int>)
AbelianGroupCategory<T>	T& operator-=(const T&)	T& repeat(int) T& negate() T& ::operator-(const T&) T& ::operator-(const T& , const T&)

Table 16.4 Categories for Single-Set Algebraic Structures with an Additive Composition Law. Each category provides the functions listed for it and for the categories listed above it in the table. Functions preceded by the scope operator :: are global; others are member functions.

an unrestricted integer argument, int, but calls the member function pow(), which restricts its integer argument.

The overall scheme is to provide implementation inheritance for the derived class—the category-provides functions—that in turn relies on name commonality in the derived class—the user-must-define functions. The category-provides functions do not know anything about the type T except that it has the user-must-define functions and a copy constructor. For example, the category has no information on what multiply means for class T beyond the call to operator*=(). Thus it is not implementation so much as relationships between functions that is inherited.

The single-set categories with an additive (Abelian) composition law are listed in Table 16.4. The functions are all the additive analogs of the functions in the multiplicative categories, except for the unary negation operator; its analog is the global inverse() function since no reasonable operator exists for unary inversion. The function repeat() means repeated addition (e.g., a.repeat(3) is a+a+a), the additive analog of pow().

16.3.2 Subdomain Classes

Before discussing the implementation of the algebra categories proper, we describe the Positive<T> and NonNegative<T> classes that are used as argument types for the repeated composition functions pow() and repeat(). These types are built on the more general SubDomain<Domain, Predicate>, a class template for types

that can hold those members of Domain that satisfy a predicate specified by a static member of Predicate.

A SubDomain<Domain, Predicate> object is created from a Domain object; creation fails if the Domain object fails to satisfy a predicate:

Algebra/SubDomain.h

```
template<class Domain, class Predicate>
class SubDomain {
public:
    SubDomain(const Domain& x) :
        value(x) {
        if ( !Predicate::isSatisfied(x) ){
          throw SubDomainErr<Domain>(value, Predicate::description());
        }
    }

    operator Domain() const { return value; }
private:
    Domain value;
};
```

Once created, SubDomain<Domain, Predicate> objects act like Domain objects through the conversion operator. The Predicate class must supply two static member functions: isSatisfied() and description(). The isSatisfied() function takes a Domain object and returns true if the predicate is satisfied; description() returns a String that can be used to construct a message if the predicate is not satisfied.

Classes like Positive<T> are built from a predicate class

Algebra/Positive.h

```
template<class T>
class PositivePredicate {
public:
    static Boolean isSatisfied(const T& x) { return x > 0; }
    static String  description()           { return "is not positive"; }
};
```

and a combination of inheritance and template expansion:

Algebra/Positive.h

```
template<class T>
class Positive :
    public SubDomain<T, PositivePredicate<T> > {
public:
    Positive(const T& x) : SubDomain<T, PositivePredicate<T> >(x) {}
};
```

The code for NonNegative is similar and is not shown.

16.3.3 Semigroup Categories

Semigroups have a multiplicative composition law; therefore SemiGroupCategory<T> requires its target classes to provide an operator*=() member function that multiplies the object for which the function is called by its argument. The category then provides multiplication of two objects and repeated composition:

Algebra/SemiGroupCategory.h

```
template<class T>
class SemiGroupCategory {
public:
    // User Must Define: T& operator*=(const T&);
    friend T operator*(const T& lhs, const T& rhs) { T temp(lhs); return temp *= rhs; }
    friend T pow(const T& x, int n)                { T temp(x); return temp.pow(n);   }
    inline T& pow(Positive<int> n);

    static void compositionLaw(T& lhs, const T& rhs) { lhs *= rhs; }
};
```

Despite the apparent simplicity of this class, its design is somewhat subtle. Consider the symmetric binary multiplication operator. As we discussed in Section 6.5, a symmetric binary operator should be declared as a global function so that conversions are applied to the operator's operands. However, we also know from Section 12.6 that we can't simply provide a global function template like this:

ch16/global-function-template.C

```
template<class T>
T operator*(const T& lhs, const T& rhs) {
    T temp(lhs);
    return temp *= rhs;
}
```

This template would supply a * operator for all possible classes T, with no way to intercept and customize the definition for any specific class T. One way to restrict the template's applicability is to make its arguments references to the category instead of references to T:

ch16/global-category-template.C

```
template<class T>
T operator*(const SemiGroupCategory<T>& lhs, const SemiGroupCategory<T>& rhs) {
    T temp((const T&) lhs);
    return temp *= (const T&) rhs;
}
```

Although this technique does restrict the applicability of the function template, it also inhibits desirable conversions. For example, if we write the expression 2*c, where c is a ComplexFloat, we would expect the int 2 to be converted to a ComplexFloat

and then to be multiplied by c. The preceding function template will not match this expression because only trivial conversions are applied when matching function templates (cf. Section 5.6).

Both restriction and conversion can be obtained by declaring and defining the global function inside of the class template definition, as in SemiGroupCategory<T>. When the class template is expanded for a particular T, the global function is declared only for arguments of type T. The global function's definition must appear in the class definition; to do otherwise would defeat the restriction we seek. This technique has the disadvantage of requiring the function definition to be inline. We judge this to be a small price to pay for simultaneously obtaining both restriction and conversion.

M §14.7c

A class is put in a function-structure category by deriving it from the function-structure class template, setting the template argument to the derived class: ComplexFloat is in SemiGroupCategory<ComplexFloat> by deriving it, through other base classes, from SemiGroupCategory<ComplexFloat>. The derived class, here ComplexFloat, supplies the user-must-define member function operator*=():

Algebra/ComplexFloat.h

```
inline ComplexFloat& ComplexFloat::operator*=(const ComplexFloat& rhs) {
    float t = real_part * rhs.real_part - imag_part * rhs.imag_part;
    imag_part = real_part * rhs.imag_part + imag_part * rhs.real_part;
    real_part = t;
    return *this;
}
```

The member and global functions provided by the category are then defined in terms of this member, as shown in the definition of SemiGroupCategory<T>.

All of the symmetric binary operators in the algebraic structure categories are defined on the same pattern: Declare a temporary initialized by the left operand; update it by calling the asymmetric binary operator function supplied by the target class; and return the temporary object. See Notes and Comments 16.3. For example, here is the definition of AbelianSemiGroupCategory<T>:

Algebra/AbelianSemiGroupCategory.h

```
template<class T>
class AbelianSemiGroupCategory {
public:
    // User Must Define: T& operator+=(const T&);
    friend T operator+(const T& lhs, const T& rhs) { T temp(lhs); return temp += rhs; }
    friend T repeat(const T& x, int n)             { T temp(x); return temp.repeat(n); }
    inline T& repeat(Positive<int> n);

    static void compositionLaw(T& lhs, const T& rhs) { lhs += rhs; }
};
```

Observe the parallel with the definition of SemiGroupCategory<T>.

The pow() and repeat() member functions also deserve mention for their implementation technique. These functions implement repeated composition. We use name commonality and templates to express a common implementation of repeated composition via a static member of the class template SemiGroupCommon<T, Structure>:

Algebra/SemiGroupCommon.c

```
template<class T, class Structure>
T& SemiGroupCommon<T, Structure>::repeatedComposition(T& x, Positive<int> n) {
    if (n > 1) {
        T t = x;
        int m = n;
        while (--m) Structure::compositionLaw(x, t);
    }
    return x;
}
```

The pow() and repeat() functions then call the repeatedComposition() static member function of the appropriate class by supplying template arguments:

Algebra/SemiGroupCategory.h

```
template<class T>
inline T& SemiGroupCategory<T>::pow(Positive<int> n) {
    return SemiGroupCommon<T, SemiGroupCategory<T> >::
                                        repeatedComposition((T&) *this, n);
}
```

Algebra/AbelianSemiGroupCategory.h

```
template<class T>
inline T& AbelianSemiGroupCategory<T>::repeat(Positive<int> n) {
    return SemiGroupCommon<T, AbelianSemiGroupCategory<T> >::
                                        repeatedComposition((T&) *this, n);
}
```

The static member compositionLaw() appears in both SemiGroupCategory<T> and AbelianSemiGroupCategory<T>, each defined to call the target class member function that defines the corresponding composition law, operator*=() for SemiGroupCategory<T> and operator+=() for AbelianSemiGroupCategory<T>. The function-structure category scheme ensures the safety of the casts.

The class SemiGroupCommon<T, Structure> is in some sense superfluous: Its only purpose is to allow us to specify a function template in which the template arguments do not appear in the function's argument list. This is a useful technique:

- Simulate function templates having template arguments that don't appear in the function's argument list by a class template with a static member function.

See also Notes and Comments 16.4.

We might have written repeatedComposition() more simply as a global function template taking a pointer to the composition law function. However, using name commonality to pass the function allows the compiler to inline the composition law, an important efficiency advantage for classes with simple composition laws.

- Where efficiency is paramount, use name commonality and templates to—in effect—pass functions by name at compile time.

16.3.4 Other Categories with One Composition Law

Monoids build on semigroups, adding an identity element expressed as an additional user-must-define function, setToOne() for monoids and setToZero() for Abelian monoids. Here is the class definition for MonoidCategory<T>:

Algebra/MonoidCategory.h

```
template<class T>
class MonoidCategory : public SemiGroupCategory<T> {
public:
    // User Must Define: T& setToOne();
    inline T& pow(NonNegative<int> n);

    static void identityElement(T& x) { x.setToOne(); }
};
```

AbelianMonoidCategory<T> is analogous. Semigroup behavior is provided via public derivation. The pow() and repeat() member functions provided in the monoid categories hide those provided in the semigroup categories; the monoid versions (not shown) take NonNegative<int> arguments, allowing repetition counts of zero. Since the pow() and repeat() global functions provided by the semigroup categories call the corresponding member functions for T, the global functions invoke the corresponding member function from the most derived category.

The monoid version of repeatedComposition() uses a more efficient algorithm that uses the identity element to start [67, page 442]:

Algebra/MonoidCommon.c

```
template<class T, class Structure>
T& MonoidCommon<T, Structure>::repeatedComposition(T& x, NonNegative<int> n) {
    // Raise x to the n-th power using the right-to-left binary method
    // of algorithm A, Knuth, The Art of Computer Programming: Seminumerical
    // Algorithms, volume 2, 2nd edition, p. 442. The variables N, Y,
    // and Z correspond to the variables of the same name in his book.
```

```
    int N = n;
    T& Y = x;    // use Knuth's name for the output variable
    T Z = x;
    Structure::identityElement(Y);

    while (1) {
        int halfN = N / 2;
        if (halfN + halfN != N) Structure::compositionLaw(Y, Z);
        N = halfN;
        if (N == 0 ) return Y;
        Structure::compositionLaw(Z, Z);
    }
}
```

Groups build on monoids, adding an inverse expressed by an additional user-must-define function, operator/=() for groups and operator-=() for Abelian groups:

ch16/GroupCategory.h

```
template<class T>
class GroupCategory : public MonoidCategory<T> {
public:
    // User Must Define: T& operator/=(const T&);
    inline T& pow(int n);
    inline T& invert();

    friend T operator/(const T& lhs, const T& rhs) { T temp(lhs); return temp /= rhs; }
    friend T inverse(const T& x)                   { T temp(x); return temp.invert();   }

    static void composeWithInverse(T& lhs, const T& rhs) { lhs /= rhs; }
    static void selfInversion(T& x)                      { x.invert();   }
};
```

The invert() member function sets the object for which it is called to the object's inverse. It is implemented in terms of the monoid identity and operator/=():

Algebra/GroupCategory.h

```
template<class T>
inline T& GroupCategory<T>::invert() {
    T& self = (T&) *this;
    T temp(self);
    self.setToOne();
    return self /= temp;
}
```

The inverse() global function is then implemented in terms of invert(), as shown in the definition of GroupCategory<T>. This structure reflects our arbitrary decision to have the target class specify inversion via the division operator instead of supplying an invert() member directly. For some objects, in-place inversion can be implemented much more efficiently than our code; those classes should provide their own invert() member, which will hide the one provided by GroupCategory<T>.

The Abelian group category is analogous, except that the unary negation operator is used for the analog of inverse():

Algebra/AbelianGroupCategory.h

```
template<class T>
class AbelianGroupCategory : public AbelianMonoidCategory<T> {
public:
    // User Must Define: T& operator-=(const T&);
    inline T& repeat(int n);
    inline T& negate();

    friend T operator-(const T& lhs, const T& rhs) { T temp(lhs); return temp -= rhs; }
    friend T operator-(const T& x)                 { T temp(x); return temp.negate();  }

    static void composeWithInverse(T& lhs, const T& rhs) { lhs -= rhs; }
    static void selfInversion(T& x)                      { x.negate();  }
};
```

The group categories extend the domain of the pow() and repeat() members to include all int objects. Again we use a common implementation of repeatedComposition() that exploits name commonality:

Algebra/GroupCommon.c

```
template<class T, class Structure>
T& GroupCommon<T, Structure>::repeatedComposition(T& x, int n) {
    if (n < 0) {
        Structure::selfInversion(x);
        n = -n;
    }
    return MonoidCommon<T, Structure>::repeatedComposition(x, n);
}
```

16.3.5 Categories for Structures with Two Composition Laws

Combining an Abelian and a non-Abelian single-law structure creates a one-set algebraic structure with two composition laws. In particular, as summarized in Table 16.1, rings, rings with unit, and fields are all combinations of an Abelian group and one of the multiplicative structures. This is reflected in the class DAG shown in Fig. 16.2.

RingCategory<T> joins an Abelian group with a semigroup to create an algebraic structure with two composition laws:

Algebra/RingCategory.h

```
template<class T>
class RingCategory :
    public AbelianGroupCategory<T>,
    public SemiGroupCategory<T> {
};
```

Likewise RingWithUnitCategory<T> joins an Abelian group with a monoid, adding prefix and postfix ++ and -- operators:

Algebra/RingWithUnitCategory.h

```
template<class T>
class RingWithUnitCategory :
    public AbelianGroupCategory<T>,
    public MonoidCategory<T> {
public:
    T& operator++()    { T one((T&)*this); one.setToOne(); return (T&)*this += one; }
    T  operator++(int) { T orig((T&)*this); ++*this; return orig;                   }
    T& operator--()    { T one((T&)*this); one.setToOne(); return (T&)*this -= one; }
    T  operator--(int) { T orig((T&)*this); --*this; return orig;                   }
};
```

These are provided for consistency with C++'s definitions of these operators for built-in types: The prefix ++x is defined for built-in types to be equivalent to x += 1 and analogously for prefix --. Our definitions provide the same relationships for the target class.

Al

The field category combines Abelian and non-Abelian groups, replicating the definitions for the ++ and -- operators:

Algebra/FieldCategory.h

```
template<class T>
class FieldCategory :
    public AbelianGroupCategory<T>,
    public GroupCategory<T> {
public:
        T& operator++()    { T one((T&)*this); one.setToOne(); return (T&)*this += one; }
        T  operator++(int) { T orig((T&)*this); ++*this; return orig;                   }
        T& operator--()    { T one((T&)*this); one.setToOne(); return (T&)*this -= one; }
        T  operator--(int) { T orig((T&)*this); --*this; return orig;                   }
};
```

We chose to replicate these operator definitions instead of deriving FieldCategory<T> from RingWithUnitCategory<T> to avoid the need for virtual derivation and its associated overhead.

The categories for two-law structures require no new user-must-define functions beyond those in their respective base classes.

16.4 Categories for Two-Set Structures

We now consider algebraic structures with two sets, each with their own algebraic structure, and an external composition law that relates the two sets. To exploit intuition, we call the two sets vectors and scalars, denoted V and S respectively, and call the external composition law scalar multiplication. The class DAG for these algebraic structures is shown in Fig. 16.3.

The category ExternalScalarsCategory<V, S> defines multiplication of vectors by scalars, given a user-must-define member function in V that specifies the meaning of multiplying a vector by a scalar:

Algebra/ExternalScalarsCategory.h

```
template<class V, class S>
class ExternalScalarsCategory {
public:
    // User Must Define: V& operator*=(const S&);
    friend V operator*(const S& s, const V& v) { V temp(v); return temp *= s; }
    friend V operator*(const V& v, const S& s) { V temp(v); return temp *= s; }
};
```

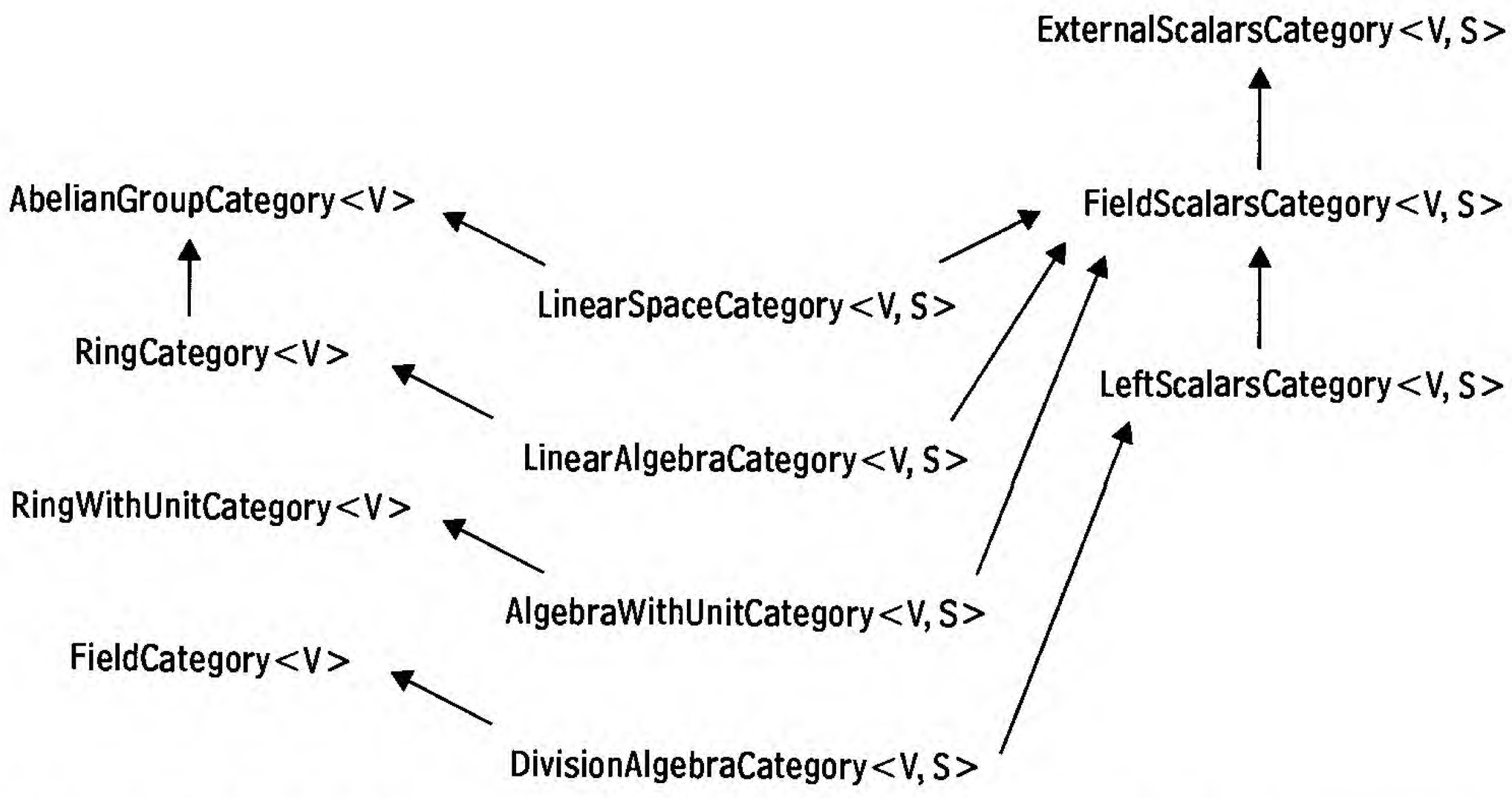


Figure 16.3 Class DAG for Two-Set Algebraic Structures. V is the class given vector structure and S is the set given scalar structure.

When the scalars have a multiplicative inverse, as when the scalars form a field, we can add division by scalars:

Algebra/FieldScalarsCategory.h

```
template<class V, class S>
class FieldScalarsCategory :
    public ExternalScalarsCategory<V, S> {
public:
    // User Must Define: V& operator/=(const S& s)
    friend V operator/(const V& v, const S& s) { V temp(v); return temp /= s; }
};
```

Finally, when the vectors have a multiplicative inverse, we can add left division of scalars by vectors:

Algebra/LeftScalarsCategory.h

```
template<class V, class S>
class LeftScalarsCategory :
    public FieldScalarsCategory<V,S> {
public:
    friend V operator/(const S& s, const V& v) { V temp(inverse(v)); return temp *= s; }
};
```

Various combinations of one-set algebraic categories with the preceding base classes create new structures. For example, DivisionAlgebraCategory<V, S> is defined like this:

Algebra/DivisionAlgebraCategory.h

```
template<class V, class S>
class DivisionAlgebraCategory :
    public FieldCategory<V>,
    public LeftScalarsCategory<V,S> {
};
```

We'll see an example use of a linear space in the next section and of a division algebra in Section 16.6.

16.5 Example: Dimensional Analysis

Engineering and scientific computation often involves physical quantities with units. We are taught dimensional analysis in our elementary science courses but generally do not carry it through to our programs. In this section, we use the algebraic structure categories to develop facilities for dimensional analysis, for checking formulas for consistency in physical units. All errors in dimensionality result in an error at compile time. For example, an attempt to assign a velocity to a mass will fail to compile. There is no runtime space or time overhead to this

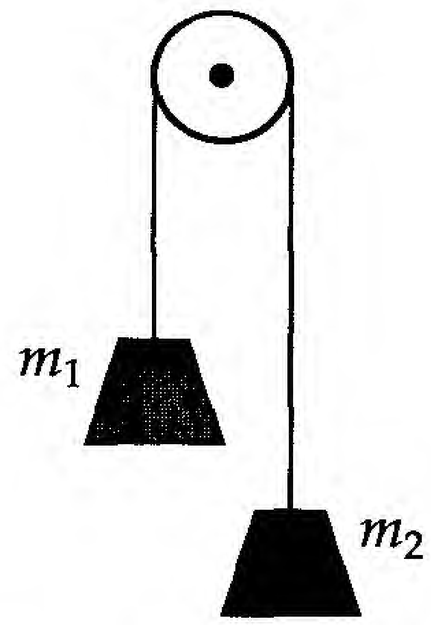


Figure 16.4 Atwood's Machine. Two unequal masses suspended by a string from a massless, frictionless pulley.

checking beyond the cost of an inline function that accesses a data member of an object.

We illustrate the idea using a simple problem from high school physics. Atwood's machine [56, Section 5-10] consists of two unequal masses suspended by a string from a massless, frictionless pulley, as illustrated in Fig. 16.4. The upward acceleration a of mass m_1 is given by

$$a = \frac{m_2 - m_1}{m_2 + m_1} g \tag{16.1}$$

and the tension t in the string is given by

$$t = \frac{2m_1 m_2}{m_1 + m_2} g, \tag{16.2}$$

where g is the magnitude of the acceleration due to gravity. These simple formulas are incorporated into the following class for Atwood's machine:

ch16/AtwoodsMachine.C

```
class AtwoodsMachine :
    private SIConstants<double> {
public:
    AtwoodsMachine(Mass m1, Mass m2) : the_m1(m1), the_m2(m2) {}
    Acceleration m1Accel() const {
        return g * (the_m2 - the_m1) / (the_m2 + the_m1);
    }
    Force tension() const {
        return g * (2 * the_m1 * the_m2) / (the_m1 + the_m2);
    }
```

```
private:
    Mass the_m1;
    Mass the_m2;
};
```

Deriving from the SIConstants<T> base class (shown on page 498) makes available various types such as Mass, Force, and Acceleration, each able to hold a value of type T, and constants such as g, kilogram, and newton. These constants are all drawn from the *Système International* (SI) system of units.

Using the types Mass, Force, and so on documents the code and implements dimensional analysis. The physical formulas for acceleration and tension are checked by the C++ compiler for dimensional consistency. For example, the units of the object returned by the function tension() must be the units of the object Force or the code will not compile. The C++ type system is being used to reduce logic errors from inconsistent or mistyped expressions.

Placing short, mnemonic names for constants as static members of base classes like SIConstants prevents name conflicts. Functions not in the scope of these base classes would have to the use scope resolution operator, SIConstants<T>::, to access the constants. To make the code both robust and easy to read, we can put functions in the scope of the base class by derivation:

ch16/AtwoodsMachine.C

```
class DemoAtwoodsMachine :
    private BritishConstants<double> {
public:
    static void demo();
};
```

The class BritishConstants<T> (not shown) derives from SIConstants<T> (page 498) and adds constants in the British system of units such as slug, foot, and pound. The function DemoAtwoodsMachine::demo() is like a global function, but the constants are accessible. Inside of it the AtwoodsMachine class can be used like this:

ch16/AtwoodsMachine.C

```
AtwoodsMachine machine1(1 * kilogram, 2 * kilogram);
cout << machine1.m1Accel() << " " << machine1.tension() << endl;
```

The masses are specified in SI units and the results are printed with the units specified.

The masses can be specified in British units like this:

ch16/AtwoodsMachine.C

```
AtwoodsMachine machine2(1 * slug, 2 * slug);
cout << machine2.m1Accel() << " " << machine2.tension() << endl;
```

Since all computation with physical quantities is done in SI units, the results are printed in SI units. It is easy, however, to convert to and from other unit systems

using the convertTo() member function. For example, the results can be printed in British units by supplying the appropriate constants:

ch16/AtwoodsMachine.

```
cout << machine2.m1Accel().convertTo(ft_per_sec2) << " "
     << machine2.tension().convertTo(pound) << endl;
```

When run, these three code segments produce the following output:

```
3.26888 L/T^2 13.0755 ML/T^2
3.26888 L/T^2 190.772 ML/T^2
10.7247 L/T^2 42.8894 ML/T^2
```

To see how the dimensional analysis can be done by the C++ compiler through the use of the abstract algebra categories, we first review dimensional analysis. Dimensional analysis works with physical quantities with fixed units. There are seven fundamental units in the SI system: mass, length, time, charge, temperature, intensity, and angle. These appear raised to integral powers in all other units. For example, denoting mass by M, length by L, and time by T, acceleration has units L^1T^{-2}, written more conveniently as L/T^2, and force has units $M^1L^1T^{-2}$, written more conveniently as ML/T^2.

The units for algebraic expressions of physical quantities are evaluated as follows: Adding (subtracting) two physical quantities is only possible if they have the same units and the result has those units; the units of the result of multiplying (dividing) two physical quantities are obtained by adding (subtracting) corresponding exponents. For example, the right side of Eq. (16.2) has units

$$\frac{MM}{M+M}\frac{L}{T^2}.$$

Evaluating the numerator and denominator yields physical quantities with units M^2L and MT^2, respectively; dividing yields a physical quantity with units ML/T^2. These are the units of force, which match the units of the formula's left side.

We need a class for physical quantities that represents the units, like ML/T^2, with a value. The units need to be known to the compiler so that it can check that the expressions we write have correct units. This implies that the units must be part of a type, not stored as member data (i.e., we need a separate type for each possible unit). We achieve this with a class template, where the exponents of the fundamental units are template arguments. We also want our physical quantities to have an appropriate algebraic structure.

To see which structure is appropriate, let's consider all physical quantities with the same units to be the set for an algebraic structure and use Fig. 16.1 to decide which structure. We can add two such quantities, but multiplying two quantities yields a quantity with different units. For example, adding two lengths

yields a length, but multiplying two lengths yields an area. Therefore we follow the additive branch of Fig. 16.1. The physical quantity with value zero is the identity, and negating the value yields an additive inverse. Therefore we have at least an Abelian group. We want to be able to multiply a physical quantity by a dimensionless number: This is external composition, so we follow that branch of the decision tree. Since there is no internal multiplicative law, we stop at a linear space.

Having decided that the units will be part of the type representing physical quantities and that the type should have the structure of a linear space, we can write the class template:

Units/Physical.h

```
template<class T, int m, int l, int t, int q, int k, int i, int a>
class Physical :
    public LinearSpaceCategory< Physical<T, m, l, t, q, k, i, a>, T > {
public:
        T& value() { return val; }
    const T& value() const { return val; }
private:
    typedef Physical<T, m, l, t, q, k, i, a> Phys;    // Shorthand
    Physical() : val(1.0){}   // A unit
public:
    static Phys unit() { return Phys(); }
    Phys convertTo(const Phys& dest_unit) const { return val / dest_unit.val * unit(); }

    // LinearSpaceCategory "user must define" functions
    Phys& operator+=(const Phys& rhs) { val += rhs.val; return *this; }
    Phys& operator-=(const Phys& rhs) { val -= rhs.val; return *this; }
    Phys& setToZero()                 { val = T(0); return *this;     }
    Phys& operator*=(T s)             { val *= s; return *this;       }
    Phys& operator/=(T s)             { val /= s; return *this;       }
private:
    T val;
};
```

No constructor taking a single `T` argument is provided: it would act as a conversion allowing assignment of a pure dimensionless quantity to a physical value with dimensions. The static function `unit()` exists to create unit value physical quantities with dimensions. The template argument `T` specifies both the type of the value stored in the physical quantity and the type of the dimensionless scalars. Typically `float` or `double` is used for `T`. The remaining template arguments specify the exponents for the units. Thus `Physical<double, 0, 1, 0, 0, 0, 0, 0>` is a class that represents physical quantities with units *L*, length.

We avoid writing such awkward class names with typedefs:

Units/FundamentalUnits.h

```
template<class T>
class FundamentalUnits {
public:
    typedef Physical<T, 1, 0, 0, 0, 0, 0, 0> Mass;
    typedef Physical<T, 0, 1, 0, 0, 0, 0, 0> Length;
    typedef Physical<T, 0, 0, 1, 0, 0, 0, 0> Time;
    typedef Physical<T, 0, 0, 0, 1, 0, 0, 0> Charge;
    typedef Physical<T, 0, 0, 0, 0, 1, 0, 0> Temperature;
    typedef Physical<T, 0, 0, 0, 0, 0, 1, 0> Intensity;
    typedef Physical<T, 0, 0, 0, 0, 0, 0, 1> Angle;
};
```

It is also convenient to define various derived units:

Units/DerivedUnits.h

```
template<class T>
class DerivedUnits :
    public FundamentalUnits<T> {
public:
    typedef Physical<T, 0, 1, -1, 0, 0, 0, 0> Velocity;
    typedef Physical<T, 0, 1, -2, 0, 0, 0, 0> Acceleration;
    typedef Physical<T, 1, 1, -2, 0, 0, 0, 0> Force;
    typedef Physical<T, 1, 2, -2, 0, 0, 0, 0> Energy;
    // ...
};
```

Physical constants can now be defined as static class members:

Units/SIConstants.h

```
template<class T>
class SIConstants :
    public DerivedUnits<T> {
public:
    static const Mass kilogram;
    static const Length meter;
    static const Force newton;
    static const Time second;
    static const Acceleration meter_per_sec2;
    static const Acceleration g;
    // ...
};
```

These constants are defined in terms of the Système International (SI) units, standardized MKS (meter-kilogram-seconds) units:

Units/SIConstants.c

```
template<class T> const SIConstants<T>::Mass
    SIConstants<T>::kilogram = SIConstants<T>::Mass::unit();
template<class T> const SIConstants<T>::Length
    SIConstants<T>::meter = SIConstants<T>::Length::unit();
template<class T> const SIConstants<T>::Force
    SIConstants<T>::newton = SIConstants<T>::Force::unit();
template<class T> const SIConstants<T>::Time
    SIConstants<T>::second = SIConstants<T>::Time::unit();
template<class T> const SIConstants<T>::Acceleration
    SIConstants<T>::meter_per_sec2 = SIConstants<T>::Acceleration::unit();
template<class T> const SIConstants<T>::Acceleration
    SIConstants<T>::g = 9.80665 * SIConstants<T>::Acceleration::unit();
```

On initialization the values on the right side will be converted to the precision of the type T. To avoid errors from out-of-order static initializations, a call to the function unit() is used in these initializers rather than expressions involving static variables. Local variables can use more natural expressions. For example, the gravitational constant could be written, in a class derived from SIConstant<T>, as Acceleration g = 9.80665 * meter / (second * second).

Referring back to the AtwoodsMachine class definition on page 494, it is derived from SIConstants<double>, which makes available units like Mass and Acceleration and constants like the gravitational constant g, with values represented using double. Derivation from SIConstants<double> adds no member data; it only injects some names for constants into the scope of the AtwoodsMachine class.

Thus far we have seen how to represent physical quantities with a class template. Each class produced by this template is endowed with the algebraic structure of a linear space, meaning that we can add and subtract physical quantities of like units and multiply and divide them by dimensionless quantities. We still must provide multiplication and division of two physical quantities with arbitrary units. For this we use function templates:

Units/Physical.h

```
template<class T, int m1, int l1, int t1, int q1, int k1, int i1, int a1,
                  int m2, int l2, int t2, int q2, int k2, int i2, int a2>
Physical<T, m1+m2, l1+l2, t1+t2, q1+q2, k1+k2, i1+i2, a1+a2>
operator*(const Physical<T, m1, l1, t1, q1, k1, i1, a1>& lhs,
          const Physical<T, m2, l2, t2, q2, k2, i2, a2>& rhs
         ) {
    return lhs.value() * rhs.value() *
           Physical<T, m1+m2, l1+l2, t1+t2, q1+q2, k1+k2, i1+i2, a1+a2>::unit();
}
template<class T, int m1, int l1, int t1, int q1, int k1, int i1, int a1,
                  int m2, int l2, int t2, int q2, int k2, int i2, int a2>
Physical<T, m1-m2, l1-l2, t1-t2, q1-q2, k1-k2, i1-i2, a1-a2>
```

```
operator/(const Physical<T, m1, l1, t1, q1, k1, i1, a1>& lhs,
          const Physical<T, m2, l2, t2, q2, k2, i2, a2>& rhs
         ) {
    return lhs.value() / rhs.value() *
           Physical<T, m1-m2, l1-l2, t1-t2, q1-q2, k1-k2, i1-i2, a1-a2>::unit();
}
```

Now when we multiply (divide) two physical quantities, the result is a physical quantity of a different type, with the exponents of the result type set as the sum (difference) of the exponents of the multiplicands. A new type is created for each unit used in a program. Although there is potentially a huge number of different types, in practice the number will be small.

An attempt to add or subtract two physical quantities with different units fails at compile time because addition and subtraction operators are only defined between physical quantities with the same units. Attempts to assign the results of the product or quotient of physical quantities to variables without matching units will also generate compile errors.

16.6 Example: Arrays with Arithmetic

For our second example using the algebraic structure categories, we look at a way to add arithmetic operators to our array system from Chapter 13. Two equally useful but separate algebras dominate scientific and engineering programming. On the one hand are the vectors and matrices of linear algebra, with multiplication of two-dimensional arrays following the rules of matrix multiplication. On the other hand are two-dimensional arrays with multiplication distributed across the arrays element by element. Rather than follow one path or the other, we extract the common features of both approaches and then create both the linear algebra and distributing systems, applying the one appropriate to each specific problem. We will illustrate the linear algebra approach in Chapter 18; here we develop a distributing version.

Our aim is to start with any one of our array classes and endow it with an element-by-element algebraic structure. A simple example would be to endow an array class having numeric elements with the structure of an Abelian semigroup, meaning that we could add two such arrays and get as the sum an array whose elements are the sum of the corresponding elements of the two addend arrays. Assuming the class is called AdditiveArray, we could write the following code:

ch16/tASGArray.C

```
AdditiveArray a(3);
a(0) = 2; a(1) = 3; a(2) = 4;
AdditiveArray b(3);
b(0) = 5; b(1) = 6; b(2) = 7;
cout << a + b << endl;    // Prints: [7, 9, 11]
```

The class itself can be defined like this:

ch16/tASGArray.C

```
class AdditiveArray :
    public ConcreteFormedArray1d<double>,
    public AbelianSemiGroupCategory<AdditiveArray> {
public:
    AdditiveArray(Subscript n) : ConcreteFormedArray1d<double>(n) {}
    AdditiveArray(const AdditiveArray& a) : ConcreteFormedArray1d<double>(a) {}
    // ... other constructors and assignment operators

    // User must define for Abelian semigroup category
    AdditiveArray& operator+=(const AdditiveArray& rhs);
};
```

It is derived publicly from a concrete array to give it its array behavior, and from the Abelian semigroup category to give it its algebraic behavior. The user-must-define function for the Abelian semigroup category iterates over its two arguments, adding corresponding elements:

ch16/tASGArray.C

```
AdditiveArray& AdditiveArray::operator+=(const AdditiveArray& rhs) {
    if (numElts() != rhs.numElts()) throw "Mismatched sizes";
    IteratorType i(*this);
    for (BrowserType j(rhs); j.more(); i.advance(), j.advance()) i.current() += j.current();
    return *this;
}
```

Writing AdditiveArray was straightforward. However, as we write various classes that endow various arrays with various algebraic structures, we'll find ourselves writing many user-must-define functions that are essentially the same. For example, the division algebra category has eight user-must-define functions, and all eight will loop over the elements of the array in the same way, applying a different operator inside the loop. Moreover, our array system supports a variety of array formats uniformly through the use of templates, so the eight user-must-define functions for column-major arrays, row-major arrays, packed arrays, and so on will all look identical. Our goal is to use the systematics of the array system and that of the algebra system together.

Building on the generality of the algebra and array systems, we create a system of array arithmetic function-structure classes parallel to the algebra system and using the array system. The purpose of these classes is to supply the user-must-define functions for particular algebraic categories. Since these classes distribute arithmetic operations over array elements, the name of each arithmetic function-structure class concatenates the word Distributing with the name of the corresponding algebraic structure.

For example, parallel to AbelianSemiGroupCategory<T> we have DistributingAbelianSemiGroup<Array, T>:

Vector/DistributingAbelianSemiGroup.h

```
template<class Array, class T>
class DistributingAbelianSemiGroup {
public:
    Array& operator+=(const Array& rhs) {
        return Distribute2<Add,Array>::over((Array&) *this, rhs);
    }
private:
    class Add { public: Add(T& lhs, const T& rhs) { lhs += rhs; } };
};
```

This class declares the operator+=() user-must-define function needed by AbelianSemiGroupCategory<T>. Any class that derives from DistributingAbelianSemiGroup<T> will inherit this function and it will be used when we add two objects in the class T. The reason for the nested class Add will become apparent at the end of the next paragraph.

The definition of the operator+=() relies on a loop-providing or distributing class, Distribute2<Op, Array>, parameterized by the operation being distributed and the kind of array being distributed over. The distributing class contains one static member function called over():

Vector/Distribute.h

```
template<class Op, class Array>
class Distribute2 {
public:
    static Array& over(Array& lhs, const Array& rhs) {
        if (lhs.numElts() != rhs.numElts()){
          throw MisMatchedSizeErr(lhs.numElts(), rhs.numElts());
        }
        Array::IteratorType i(lhs);
        Array::BrowserType j(rhs);
        for (; i.more(); i.advance(), j.advance()) Op(i.current(), j.current());
        return lhs;
    }
};
```

The parameter Op represents, for example, operator+=(). We don't want to pass a pointer to the function as this would imply runtime overhead. Thus we pass a class name to Distribute2<Op, Array>, a class whose inline constructor implements the operation to be distributed. For example, in DistributingSemiGroup<Array, T>, we call over() setting Op to Add and implement Add inline, with its constructor adding its two arguments. A good compiler can generate code that applies += directly to the elements with no extra function call.

The rest of the distributing system works in a parallel fashion. All of the user-must-define binary operations are provided by calling Distribute2<Op, Array> with an appropriate class, Op, defined within the distributing base class. Unary operations like negation are provided by calling Distribute1<Op, Array> (not shown), and distributing a scalar over the array elements is done in DistributeS<Op,Array> (not shown).

Only when we combine a scalar distributing class with a vector distributing class having a non-Abelian operation do we find a new issue: The operator*=() members from the two sides collide. Thus we must disambiguate these operators in the usual fashion, as we show for DistributingDivisionAlgebra here:

Vector/DistributingDivisionAlgebra.h

```
template<class Array, class T, class S>
class DistributingDivisionAlgebra :
    public DistributingField<Array, T>,
    public DistributingLeftScalars<Array, T, S> {
public:
    Array& operator*=(const Array& rhs) {
        return DistributingField<Array, T>::operator*=(rhs);
    }
    Array& operator/=(const Array& rhs) {
        return DistributingField<Array, T>::operator/=(rhs);
    }
    Array& operator*=(const S& rhs)      {
        return DistributingLeftScalars<Array, T, S>::operator*=(rhs);
    }
    Array& operator/=(const S& rhs)      {
        return DistributingLeftScalars<Array, T, S>::operator/=(rhs);
    }
};
```

This ambiguity did not appear in the algebraic system because these are user-must-define operators and hence do not appear in the category-defining base classes.

To see this system in action, consider implementing an arithmetic class based on compile-time fixed-size arrays and providing all of the operations supported by floating point elements. Our version is

Vector/RigidArithmetic.h

```
template<class T, Subscript n0>
class RigidArithmetic1d :
    public DivisionAlgebraCategory< RigidArithmetic1d<T, n0>, T >,
    public DistributingDivisionAlgebra<RigidArithmetic1d<T, n0>,T, T>,
    public DistributingEquivalentCategory<RigidArithmetic1d<T, n0> >,
    public ConcreteRigidArray1d<T, n0> {
```

```
    public:
        RigidArithmetic1d() : ConcreteRigidArray1d<T, n0>() {}
        RigidArithmetic1d(Subscript n) : ConcreteRigidArray1d<T, n0>(n) {}
        RigidArithmetic1d(const RigidArithmetic1d<T, n0>& a) :
            ConcreteRigidArray1d<T, n0>(a) {}

        const RigidArithmetic1d<T, n0>& operator=(const RigidArithmetic1d<T, n0>& rhs) {
            ConcreteRigidArray1d<T, n0>::operator=(rhs);
            return *this;
        }
        const RigidArithmetic1d<T, n0>& operator=(const T& rhs) {
            ConcreteRigidArray1d<T, n0>::operator=(rhs);
            return *this;
        }
};
```

As with all of the array classes, these vector classes must redefine constructors and assignment operators. Most of the other operations are defined by one of the four base classes: the array operations from `RigidArray1d<T, n0>`; the algebra operations in `DivisionAlgebraCategory`, except the user-must-define functions; and the user-must-define functions implemented with the array operations in `DistributingDivisionAlgebra` and `DistributingEquivalentCategory`. This latter class is not shown; it implements operators `==` and `!=` by elementwise operations on arrays.

`RigidArithmetic1d<T, n0>` is but one example of the wide range of possible combinations we can now support. In addition to supporting the spectrum of array formats, we can attach just the appropriate algebra to the array. Depending on the algebraic properties of the elements of the array, we can selectively provide just those operations in the distributing array that will be valid for the elements. Moreover, through a combination of hiding inherited functions and template specialization, we can selectively tune the distribution class for performance or special application.

16.7 Summary

We have summarized abstract algebra and toured its simplest forms. We then expressed some of the commonality contained in abstract algebra in C++, using the commonality to treat operators systematically and consistently. We considered examples from complex numbers with `float` storage, dimensional analysis, and distributing arithmetic over array elements.

The combination of the abstract view of mathematics in the algebraic structures and our heavy use of templates and inheritance in these examples makes this

material especially difficult. To top it off, we now ask you to abstract from what we have presented to see the kernel concept: We can express function-structure commonality in C++, like the rule that `c=a+b` means `c=a; c+=b`, using a combination of templates and inheritance that we call the function-structure category class. And if you can look even deeper, you can see that the driving force for all of the techniques we describe as categories is to capture commonality, commonality beyond the algorithmic or data structure commonality we have in FORTRAN or C.

16.8 Notes and Comments

16.1 Algebraic structures are studied in undergraduate mathematics and, increasingly, in computer science. Many texts are available, with varying degrees of sophistication. We found the treatment in [96, Part IIA] particularly helpful.

16.2 It seems impossible to define a useful exponentiation operator in C++ because no suitable operator symbol has the appropriate arity, associativity, and precedence.

16.3 Our definition of SemiGroupCategory<T>::operator*() (page 485) follows the form suggested in [44, Section 12.1.1c] to allow compilers to eliminate the copy implied by the return of temp. A good compiler can generate excellent code from this form.

16.4 The ANSI C++ committee is considering a syntax for calling function templates in which all template arguments need not appear in the function's argument list. If adopted, this syntax would eliminate the need to simulate such function templates with a static member function in a class template, as we did in the algebraic structure categories.

16.5 We first encountered the technique of using nested typedef declarations and templates to compose functions while still allowing inlining in [107, Section 8.4].

16.6 Logic dictates that RingWithUnitCategory<T> should be derived from RingCategory<T> as well as from MonoidCategory<T> and AbelianGroupCategory<T>. We decided not to do this to avoid the overhead introduced by virtual base classes.

16.7 The function templates defined in Section 16.5 for multiplying and dividing physical quantities have nontype template arguments. Nontype arguments for function templates are prohibited in the *Annotated C++ Reference Manual* [44], but allowed under a recent tentative decision by the ANSI standardization committee. The compiler we used implemented this decision. This code will not compile on a compiler that still prohibits nontype function template arguments. ARI

16.9 Exercises

16.1 Define a class for the semigroup of integers under multiplication modulo 5 such that the program

ch16/tIntMod5.C

```
int main() {
    IntMod5 a = 2;
    IntMod5 b = 4;
    cout << a * b << " " << a.pow(4) << " " << a << " " << pow(b, 3) << endl;
    return 0;
};
```

produces the output

```
3 1 1 4
```

6.2 Add prefix and postfix increment and decrement operators to the class you wrote for Exercise 16.1.

6.3 Complete the implementation of ComplexFloat (page 479). Evaluate its performance.

6.4 We put the ComplexFloat class in the field category. Complex numbers can also be multiplied and divided by real numbers. Modify ComplexFloat to capture this additional property. Discuss the advantages and disadvantages of the original class and your modified class.

6.5 The three-dimensional vectors of analytic geometry have an additive composition law (element-by-element addition), an additive identity (the zero vector), and an additive inverse (negation of the elements). There is an external composition law (multiplication by a scalar) and a second composition law (vector cross product). There is no identity element for vector cross product. In which algebraic structure categories should a three-dimensional vector class be?

6.6 A three-dimensional vector class, as described in Exercise 16.5, should make it easy to translate vector computations into C++ code. For example, given three points with position vectors **a**, **b**, and **c**, the radius r of the circle that passes through those points is given by

$$r = \frac{|\mathbf{a}-\mathbf{c}||\mathbf{b}-\mathbf{c}||\mathbf{a}-\mathbf{b}|}{2|(\mathbf{a}-\mathbf{c})\times(\mathbf{b}-\mathbf{c})|}.$$

The translation into code should look something like this:

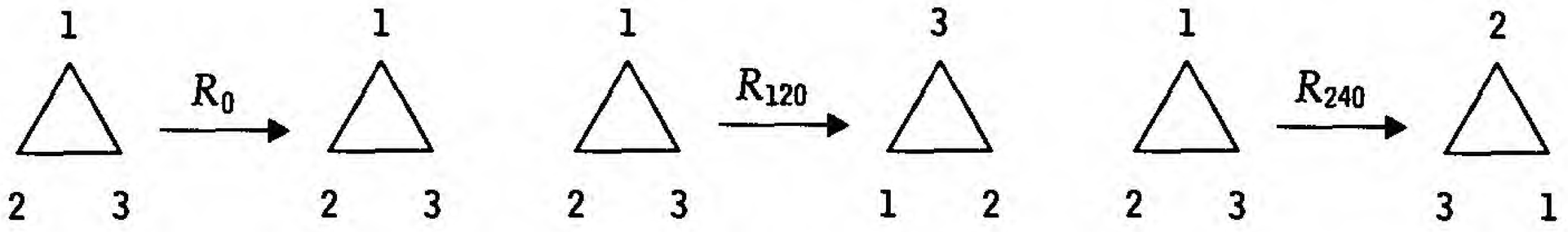
ch16/tSpaceVector.C

```
r = ( (a-c).norm() * (b-c).norm() * (a-b).norm() ) /
    ( 2.0 * ((a-c) * (b-c)).norm() );
```

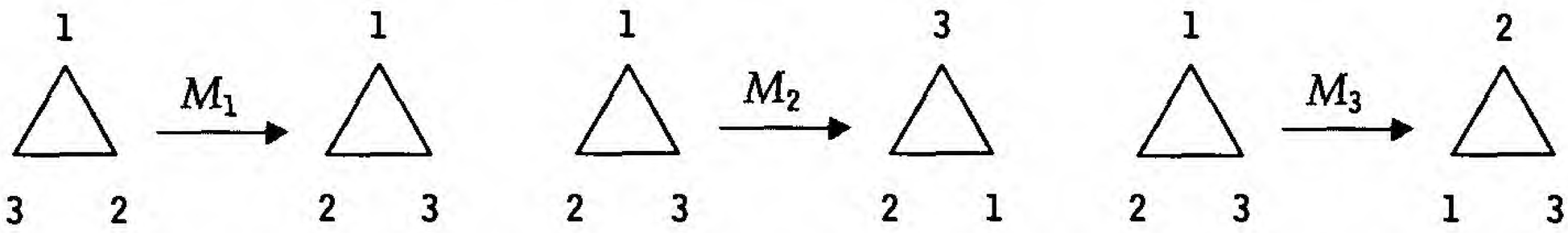
Implement a ConcreteVector3d class having this behavior using double values for the elements. Generalize the result to use generic type T for the elements. Try to create a vector of complex numbers from the last version.

6.7 Add to the dimensional analysis classes the ability to compare two physical quantities with the operators <, >, ==, !=, <=, and >=.

16.8 Algebraic structures need not be numerical. Some interesting structures use functional composition as the composition law and operators themselves as elements of the set (e.g., the symmetry group of the equilateral triangle). The elements of the set are the symmetry operations on plane triangles, the isometries of the plane (transformations that preserve length) that bring the triangle back into itself [63, Chapter 12]. These elements are the three rotations about the triangle's centroid



and the three reflections through the triangle's medians



The composition law is sequential application of the operations (e.g., $M_1 \square M_2 = R_{120}$). The composition law is defined in its entirety by the following multiplication table:

$\square$	R_0	R_{120}	R_{240}	M_1	M_2	M_3
R_0	R_0	R_{120}	R_{240}	M_1	M_2	M_3
R_{120}	R_{120}	R_{240}	R_0	M_3	M_1	M_2
R_{240}	R_{240}	R_0	R_{120}	M_2	M_3	M_1
M_1	M_1	M_2	M_3	R_0	R_{120}	R_{240}
M_2	M_2	M_3	M_1	R_{240}	R_0	R_{120}
M_3	M_3	M_1	M_2	R_{120}	R_{240}	R_0

The identity is R_0 (rotate 0 degrees), and since every row and every column contains R_0, every element has an inverse. This group is also called the *dihedral group* of degree 3, often denoted D_3. Implement D_3 as GroupD3 with a nested enumeration

ch16/tGroupD:

```
enum sym_op { R0, R120, R240, M1, M2, M3 };
```

such that the following program:

ch16/tGroupD:

```
int main() {
    cout << GroupD3(GroupD3::R240) * GroupD3(GroupD3::M1)
         << " " << pow(GroupD3(GroupD3::M2), 4) << endl;
    return 0;
}
```

produces

```
M2 R0
```

See Exercises 16.9 and 16.10 for interesting applications of dihedral groups.

16.9 Implement a Polygon class that provides functions for creating and displaying polygons. Implement additional functions that apply the isometries of the dihedral group D_3 (counter clockwise rotation of 0, 120, and 240 degrees, reflections through the lines $x = 0$, $x = \sqrt{3}y$, and $x = -\sqrt{3}y$; cf. Exercise 16.8) to Polygon instances. *Hint:* Observe from the multiplication table for D_3 that each rotation can be implemented as the product of two reflections. Where should these functions reside (i.e., in Polygon, in GroupD3, or elsewhere)?

16.10 A kaleidoscope is an instrument that contains pieces of colored glass (or plastic) loosely sandwiched between two transparent or translucent parallel flat plates. Beautiful, moving patterns are formed by changing the position of the glass pieces and viewing their images in two plane mirrors that intersect at an angle of $180/n$ degrees (n an integer). When $n = 3$, the pattern consists of the glass shapes transformed by the members of D_3. Using the classes you wrote in Exercise 16.9, implement a program that takes as input a collection of polygons contained in a 60-degree sector and draws the pattern formed by an $n = 3$ kaleidoscope. See [30, Section 2.7] and [63, Chapter 12] for more information on kaleidoscopes.

16.11 Quaternions [96, Section 4.2] are the set of 4-tuples (a, b, c, d) of real numbers with the additive composition law

$$(a, b, c, d) + (a', b', c', d') = (a + a', b + b', c + c', d + d')$$

and the multiplicative composition law

$$(a, b, c, d)(a', b', c', d') = (aa' - bb' - cc' - dd', ba' + ab' + dc' - cd',$$
$$ca' - db' + ac' + bd', da' + cb' - bc' + ad').$$

It is easy to see that the additive identity is $(0, 0, 0, 0)$, the multiplicative identity is $(1, 0, 0, 0)$, the additive inverse of (a, b, c, d) is $(-a, -b, -c, -d)$, and the multiplicative inverse of (a, b, c, d) is

$$\frac{1}{a^2 + b^2 + c^2 + d^2}(a, -b, -c, -d).$$

Quaternions, a generalization of complex numbers (set $c = d = 0$), have applications in physics and mechanics. For example, they are used in robotics to represent rotations about an axis in three-dimensional space [110]. Using the algebraic structure categories, implement a class to represent quaternions.

16.12 An alternating current $I\cos(\omega t + \varphi)$ at frequency ω can be represented by the complex number $Ie^{i\varphi}$. The representation of the sum (difference) of two currents at the same frequency ω is the sum (difference) of their currents. The product of the representations of two currents has no meaning. Define a class for representing currents at frequency ω.

16.13 A group G is a *cyclic group* if every element is a power of a fixed element $a \in G$. If G has a finite number n of elements, the group is a cyclic group of order n and $G = \{1, a, a^2, \ldots, a^{n-1}\}$. Implement a class template for cyclic groups of order n.

16.14 Let $G = \{a, b, c, \ldots\}$ and $G' = \{a', b', c', \ldots\}$ be two groups. The Cartesian product $G \times G'$ (ordered pairs of elements from G and G') is a group, called the direct product, with composition law $(a, a') \square (b, b') = (a \square b, a' \square b')$. Implement a class template for the direct product of two groups.

CHAPTER 17

Function Objects

In this chapter, we look at function objects. C++ built-in functions are not C++ objects: They cannot be copied, assigned, or altered. C++ does provide pointers to functions, and these pointers can be used to build objects. The simplest such object can be called a *function object*. Function objects associate state (member data) and behavior (class member functions) with a function but otherwise act like a function. They introduce a level of abstraction that allows us to think of and work with a trig function rather than sin() or cos() specifically. Working in terms of such abstractions, we show how function objects can be used to formulate partially constant, multiple-parameter functions and algebraic structures over sets of functions. We then explore the interaction of function objects with collections.

17.1 Function Pointers

Our first task is to explore built-in function pointers, the C++ mechanism for referring to and delaying the evaluation of functions. A *function pointer* is a pointer that can be set to refer to a function. When the pointer is dereferenced, the function referred to is called; when called, the appropriate function arguments must be supplied.

For example, we can declare a pointer to a function that takes a double argument and returns a double result like this:

ch17/tPtrSin.C

```
double (*my_trig)(double);
```

The syntax is unusual in that the name of the pointer is written inside the type specification and parentheses are used in the declaration both for grouping and to indicate function arguments. The first parentheses group the * with the variable name my_trig: my_trig is a pointer to something. Without these first parentheses, my_trig would be the name of a function that takes a double argument and returns a pointer to a double. The second parentheses indicate function arguments, as usual.

Many people find this syntax awkward, so a typedef is commonly used to provide a simple name for a function pointer type:

ch17/tPtrSin.C

```
typedef double (*TrigFncPtr)(double);
```

Although the typedef itself remains awkward, declarations of pointers to functions appear similar to declarations of other types:

ch17/tPtrSin.C

```
TrigFncPtr my_trig = sin;
```

The variable my_trig is a pointer to a function that takes a double and returns a double and is initialized to point at the sin() function. We did not have to use the address-of operator (&) to obtain a pointer to sin() because C++ allows a function name to initialize a pointer to the function, except when the name is used as the operand of the address-of or function call operators.

M §4.6

The function pointed to by a function pointer is called by supplying an argument list (possibly empty), with or without using the indirection operator (*):

ch17/tPtrSin.C

```
double trig_val;
trig_val = (*my_trig)(1.0);      // Call using indirection operator
trig_val = my_trig(2.0);         // Call without using indirection operator
```

The two forms are equivalent.

Function pointers can point at overloaded functions. Suppose sin() were overloaded like this:

ch17/tPtrSin.C

```
extern double sin(double);
extern float  sin(float);
```

The preceding assignment to my_trig would set it to point at sin(double), whereas

ch17/tPtrSin.C

```
typedef float (*FltTrigFncPtr)(float);
FltTrigFncPtr your_trig = sin;
```

initializes your_trig to point to sin(float). When a function pointer is initialized or assigned to, the argument types specified in the pointer type are used to select from among all functions of the given name. An exact match of both argument and return types is required:

1 §13.3

ch17/tPtrSin.C

```
typedef float (*IntTrigFncPtr)(int);
IntTrigFncPtr int_trig       = sin;    // WRONG: No exact match on argument types
typedef double (*MixedTrigFncPtr)(float);
MixedTrigFncPtr mixed_trig = sin;    // WRONG: Mismatch on return type
```

Since function templates act like overloaded functions, initializing or assigning a variable to the name of a function template causes the template to be expanded for the types appearing on the left-hand side. For example, using the sqr() template from page 104 and the foregoing typedef and variable, we can write

ch17/tPtrSin.C

```
TrigFncPtr sqr_as_trig = sqr;      // Expands double sqr(double).
your_trig = sqr;                   // Expands float sqr(float).
```

This kind of construct helps make code robust: A change to the typedef will change the template expansion without any manual changes to the source code using the assignment.

17.2 Member Function Pointers

The function pointers described in the previous section cannot point to a non-static member function because a non-static member can only be invoked with an instance of its class. In other words, if we want an object that can refer to any non-static member function of a class, we must say which class we are talking about. This requires that the syntax for declaring a function pointer be modified to accept a class name. Here is an example using the ComplexFloat class (page 479) from Section 16.2:

ch17/ptr-member.C

```
typedef ComplexFloat& (ComplexFloat::* CFOp)(const ComplexFloat&);
CFOp op = ComplexFloat::operator+=;
```

As before, we use a typedef for convenience. This declaration is similar to the declaration of nonmember function pointers from the preceding section except that the * in the declaration is preceded by the class name and the scope resolution operator (::). A *pointer to member* is initialized (or assigned to) by naming an appropriate member of the class; as with ordinary function pointers, argument and return types must match exactly.

To call a member function through a pointer to a member function, we must supply an instance of the proper class in addition to the arguments. The *pointer to member operator* .* is used between the class instance and the pointer to member:

ch17/ptr-member.C

```
ComplexFloat a(1, 2);
ComplexFloat b(3, 4);
cout << (a.*op)(b) << endl;
```

The parentheses are required because the function call operator has higher precedence than the pointer to member operator. Assignment can change the value of a (non-const) pointer to member:

ch17/ptr-member.C

```
op = ComplexFloat::operator-=;
cout << (a.*op)(b) << endl;
```

This time the -= operator in ComplexFloat is called.

Just as the dot operator (.) for objects and the arrow operator (->) for pointers come in a pair, the pointer to member operator (.*) has a pointer form (->*). It takes a pointer to an appropriate class object as its left operand:

ch17/ptr-member.C

```
ComplexFloat* p = &a;
cout << (p->*op)(b) << endl;
```

We shall make use of pointers to member functions in Section 17.7.

C++'s built-in function pointers provide all of the capabilities for delaying function evaluation available in the language. Nothing we introduce in the remainder of this chapter will expand the range of things that we can do with functions. Instead, our aims parallel the work of Chapter 13 on arrays and Chapter 14 on pointers: to restrict raw function pointers to make their application safer and to package them with other objects to make them easier to use.

17.3 Function Objects

Superficially, an object that provides the function call operator, operator()(), behaves like a function. The function call operator must be a member function, with parameter and return types like other member functions. To invoke this function, the object identifier is followed by arguments enclosed in parentheses. For example, a class with an operator()() member function can be defined like this:

ch17/tSimpleTrig.C

```
class SimpleTrig {
public:
    SimpleTrig(double (*f)(double)) : the_function(f) {}
    double operator()(double v) { return the_function(v); }
private:
    double (*the_function)(double);
};
```

When objects of this type are created, they can appear in expressions just like sin() might:

ch17/tSimpleTrig.C

```
SimpleTrig aTrigF(sin);      // Capture double sin(double);
double d1 = sin(2.3);        // Call sin function.
double d2 = aTrigF(2.3);     // Call operator, which then calls sin function.
```

Of course, these simple examples could well be handled with function pointers. Function objects are useful for adding something beyond functionlike behavior. We now consider storing auxiliary parameters needed for function evaluation in function objects.

17.3.1 Functions with Hidden Variables

In mathematics, it is common for a function of several independent variables to be treated as a function of one or two of the variables with the remaining variables considered parameters. For example, the *normal probability density function*

$$N(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(x-\mu)^2/\sigma^2}$$

is a function of three independent variables x, μ (mean), and σ (standard deviation), but, as is indicated by the notation $N(x;\mu,\sigma)$, it is usually treated as a function of x, with μ and σ considered fixed parameters.

This function can be represented by an object that behaves like a function and stores the mean and standard deviation:

Function/NormalDensityFunction.

```
template <class D>
class NormalDensityFunction {
public:
    NormalDensityFunction(const D& mean, const D& std_dev);
    D& mean() { return the_mean;    }
    D& width() { return the_std_dev; }
    D operator()(const D& x) const;
protected:
    D the_mean;
    D the_std_dev;
};
```

The stored parameters are used in computing the function value:

Function/NormalDensityFunction.

```
template<class D>
D NormalDensityFunction<D>::operator()(const D& x) const {
    const double recip_sqrt_2_pi = .3989423;
    const double recip_std_dev = 1.0 / the_std_dev;
    return recip_std_dev * recip_sqrt_2_pi *
                                        exp( -.5 * sqr((x - the_mean) * recip_std_dev) );
}
```

Writing this as a class template simplifies using various floating point precisions, or extending to complex or other types (see Chapter 19). Such function objects can be created in one part of a program

ch17/tNormalDensityFunction.C

```
NormalDensityFunction<double> std_normal(0.0, 1.0);
NormalDensityFunction<double> std_biased(1.0, 1.0);
```

and used as a function in another part

ch17/tNormalDensityFunction.(

```
cout << std_normal(0.0) << " " << std_normal(1.0) << endl;
cout << std_biased(0.0) << " " << std_biased(1.0) << endl;
```

to give

```
0.398942 0.241971
0.241971 0.398942
```

Using function objects to represent functions with parameters helps separate code that determines or adjusts the parameter from code that evaluates the function over its independent variable(s). We use this technique in a data modeling example in Chapter 19.

17.4 Functional Interface Categories

Client functions can use the category of classes that share a common operator()() member function declaration by creating a function object interface category:

Function/Functional.]

```
template<class Domain, class Range>
class Functional {
public:
    virtual Range operator()(const Domain&) const = 0;
    virtual Functional<Domain, Range>* clone() const = 0;    // Must be overridden!
    virtual ~Functional(){};
};
```

We use the mathematical terms Domain and Range for the function's argument and return types, respectively. Classes in this category can be used via the interface base, and they will all contain a function operator with the specified domain and range types. They are also cloneable in that they have a clone() function that can be called to copy instances of classes in the category, as is required by pointers like CloneableObjPtr<T> on page 443.

IsoFunctional<Domain> is the category of functionals with the same domain and range:

Function/Functional.]

```
template<class Domain>
class IsoFunctional :
    public virtual Functional<Domain, Domain> {
public:
    virtual IsoFunctional<Domain>* clone() const = 0;    // Must be overridden!
};
```

It allows outputs from function objects to be used as inputs to other function objects of the same type (see the next section), and it allows in-place transformations of collections of objects (see Section 17.7). We will use the function and clone() commonality of IsoFunctional<T> in the next section to create an expression tree, a dynamically created list of functions.

Functional<Domain, Range> objects can be created from built-in function pointers with a trivial wrapper:

Function/Function.h

```
template<class Domain, class Range>
class Function :
    public virtual Functional<Domain, Range> {
public:
    Function( Range (*f)(Domain) ) : pf(f)              {}
    Function()                     : pf(unsetPtrErr) {}

    const Function<Domain, Range>&
        operator=(const Function<Domain, Range>& f) { pf = f.pf; return *this; }
    const Function<Domain, Range>&
        operator=(Range (*f)(Domain))                { pf = f;    return *this; }

    virtual Range operator()(const Domain& x) const { return pf(x); }

    virtual Function<Domain, Range>*
        clone() const { return new Function<Domain, Range>(*this); }

    static Range unsetPtrErr(Domain);    // Throws UnsetPtrErr
protected:
    Range (*pf)(Domain);
};
```

Similarly, the IsoFunctional<Domain> interface can be wrapped around a built-in pointer by restricting the return type in Function<Domain, Range>:

Function/Function.h

```
template<class Domain>
class IsoFunction :
    public virtual IsoFunctional<Domain>,
    private Function<Domain, Domain> {
public:
    IsoFunction( Domain(*f)(Domain) ): Function<Domain,Domain>(f) {}
    Function<Domain, Domain>::operator();
    virtual IsoFunction<Domain>*
        clone() const { return new IsoFunction<Domain>(*this); }
};
```

Instances of classes in the IsoFunctional<Domain> category can be called through the interface:

ch17/tFunction.C

```
IsoFunction<double> aSin(sin);                    // Capture sin;
IsoFunctional<double>& aTrigF=aSin;               // Ok

extern void aFunction(const IsoFunctional<double>&);
aFunction(aSin);                                  // Will call sin() through base class.
```

Using an interface base allows objects from different classes, with different definitions for the mapping of double values to double values, to be manipulated with the same code.

17.5 Function Objects for Delayed Evaluation

Mathematical expressions like $\exp(\sin^2(x)) + \sin(x)$ are functions, with the variables like x being arguments. We can design a class of function objects that allow us to write mathematical expressions at runtime, manipulate the expressions as objects, and later evaluate these expressions. For example, we want to create objects $f = \exp(x^2)$, $g = \sin(x)$, and $h = f(g) + g$ and finally evaluate $h(2.0)$. Note that $f(g)$ means function composition: The x in $f(x)$ is replaced by a function $g(x)$, not by the value of $g(x)$ for a particular x. The function object h represents $\exp(\sin^2(x)) + \sin(x)$, and it can be evaluated for any x.

To represent mathematical expressions, we create a class with algebraic properties so we can add, subtract, and so on, and with functional properties so we can compose and evaluate functions. This class must have aspects of both interfaces and objects. Our class must have an interface aspect because it must hold variables (x), constants (2), built-in functions (exp()), and composed functions ($g(f)$): objects of different types but common interface. The value-oriented algebraic properties like addition must work on objects: Addition generates a new object from the sum of two existing objects ($h = f(g) + g$). In addition, mathematical expressions are more natural when expressed with objects rather than pointers to interfaces: We do not want to write *h = *f(*g) + *g. Thus we need a combination of interface and object properties similar to the accessor objects in Section 13.6.1.

To achieve both properties, we create a class of objects in an interface category for functional objects (IsoFunctional<Domain>) and inside these objects we hold a pointer to this interface base. We call the corresponding class FunctionalAlgebra<Domain>:

Function/FunctionalAlgebra.h

```
template<class Domain>
class FunctionalAlgebra :
    public virtual IsoFunctional<Domain>,
    public DivisionAlgebraCategory< FunctionalAlgebra<Domain>, Domain > {
public:
```

Being in the IsoFunctional<Domain> category, instances of this class behave like functions that map const Domain& objects to Domain objects; DivisionAlgebraCategory<V, S> (Section 16.4) then provides algebraic operations on these functions.

Constructors in the class provide ways of injecting objects from outside the algebra:

Function/FunctionalAlgebra

```
FunctionalAlgebra(const CountedBuiltInPtr<Domain>& d);   // Means a parameter
FunctionalAlgebra(const Domain& d);                      // Means a constant.
FunctionalAlgebra();                                     // Means a variable (x).
FunctionalAlgebra(const IsoFunctional<Domain>& fp);      // Means a function to be copied
FunctionalAlgebra(IsoFunctional<Domain>* just_newed);    // Means a function, don't copy
```

An instance constructed with no arguments represents $f(x) = x$, and an instance constructed with a T argument d represents a constant $f(x) = d$. A parameter that can change after the expression is built, but before it is evaluated, is constructed from a pointer to a shared object, a CountedBuiltInPtr<Domain>. The way these constructors work should be clearer when we examine their implementation later in this chapter.

Evaluation and composition are declared as overloaded function call operators, with a Domain argument selecting evaluation and a FunctionalAlgebra<Domain> argument selecting composition; declarations for members needed by the algebra category and declarations for built-in transcendental functions follow the function call declarations:

Function/FunctionalAlgebra.h

```
virtual Domain
    operator()(const Domain& v) const { return (*f)(v); }
CloneableObjPtr< IsoFunctional<Domain> >
    operator()(const FunctionalAlgebra<Domain>& inner) const;

    // DivisionAlgebraCategory User Must Define.
FunctionalAlgebra<Domain>& operator+=(const FunctionalAlgebra<Domain>& rhs);
FunctionalAlgebra<Domain>& operator-=(const FunctionalAlgebra<Domain>& rhs);
FunctionalAlgebra<Domain>& operator*=(const FunctionalAlgebra<Domain>& rhs);
FunctionalAlgebra<Domain>& operator/=(const FunctionalAlgebra<Domain>& rhs);
FunctionalAlgebra<Domain>& operator*=(const Domain&);
FunctionalAlgebra<Domain>& operator/=(const Domain&);
FunctionalAlgebra<Domain>& setToZero();
FunctionalAlgebra<Domain>& setToOne();
FunctionalAlgebra<Domain>& negate();
FunctionalAlgebra<Domain>& invert();
```

```
    // Transcendental functions
friend FunctionalAlgebra<Domain> exp(const FunctionalAlgebra<Domain>&);
friend FunctionalAlgebra<Domain> log(const FunctionalAlgebra<Domain>&);
friend FunctionalAlgebra<Domain> sqrt(const FunctionalAlgebra<Domain>&);
friend FunctionalAlgebra<Domain> log10(const FunctionalAlgebra<Domain>&);
friend FunctionalAlgebra<Domain> sin(const FunctionalAlgebra<Domain>&);
friend FunctionalAlgebra<Domain> cos(const FunctionalAlgebra<Domain>&);
friend FunctionalAlgebra<Domain> asin(const FunctionalAlgebra<Domain>&);
friend FunctionalAlgebra<Domain> acos(const FunctionalAlgebra<Domain>&);

    virtual FunctionalAlgebra<Domain>*
        clone() const { return new FunctionalAlgebra<Domain>(*this); }
private:
    CloneableObjPtr< IsoFunctional<Domain> > f;
};
```

The class ends with the single member datum: Each instance holds a CloneableObjPtr<T> pointer to any IsoFunctional<Domain>. Thus FunctionalAlgebra<Domain> objects control the lifetime of and can copy IsoFunctional<Domain> objects without knowing their detailed character. In this way we obtain the combination of interface and object characteristics we need.

Figure 17.1 illustrates the dual nature of FunctionalAlgebra<Domain> objects. They are objects in the interface category IsoFunctional<Domain>, but—like an interface—the exact behavior of these objects depends on the type of object they point to internally.

Let's trace through the execution of a simple example:

ch17/tSimpleFunction.C

```
FunctionalAlgebra<double> x;    // Variable by default.
FunctionalAlgebra<double> f = x + x;
cout << f(5) << endl;
```

The first statement defines an independent variable x by using the default constructor. The second statement creates an object, f, that represents the function $f(x) = x + x$, and the third statement evaluates that function at 5, producing 10 as the result. In the following paragraphs we look at the function calls triggered by this code to see how FunctionalAlgebra<Domain> works.

The first statement uses the default constructor:

Function/FunctionalAlgebra.c

```
template<class Domain>
FunctionalAlgebra<Domain>::FunctionalAlgebra() :
    f( new XFunctional<Domain>() ) {
}
```

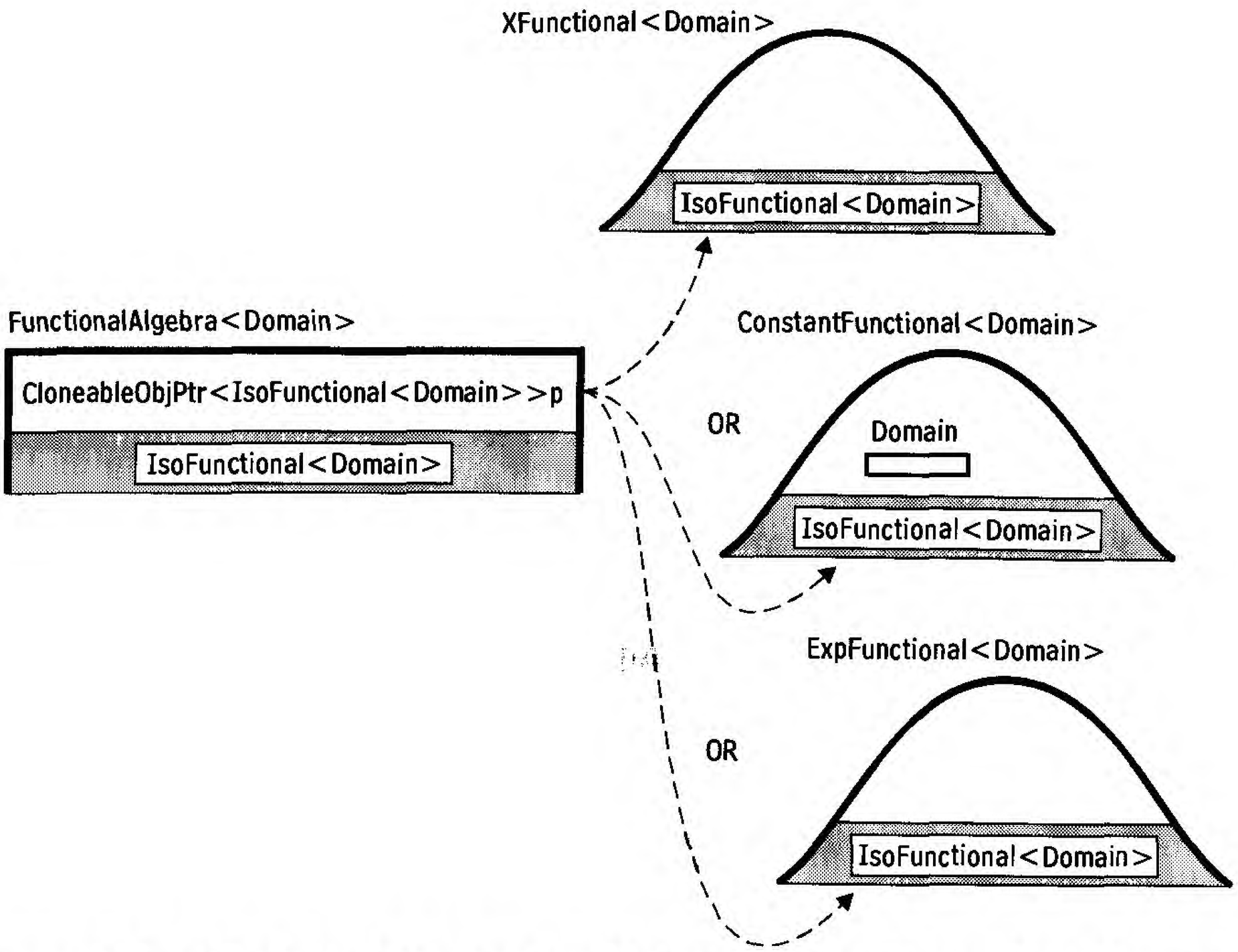


Figure 17.1 Object Sketch for FunctionalAlgebra<Domain>. These objects have an IsoFunctional<Domain> interface and they contain a pointer to an object in this category, in the manner of an accessor. The objects pointed to can be, for example, objects that represent variables (XFunctional<Domain>), constants (ConstantFunctional<Domain>), or various functions (ExpFunctional<Domain>).

The resulting object holds a pointer to an XFunctional<Domain> object; this is an IsoFunctional<Domain> functionlike object that implements $f(x) = x$:

Function/XFunctional.]

```
template<class Domain>
class XFunctional :
    public virtual IsoFunctional<Domain> {
public:
    virtual Domain operator()(const Domain& v) const { return v; }
    virtual XFunctional<Domain>*
        clone() const { return new XFunctional<Domain>(*this); }
};
```

Thus a function that returns its argument represents the variable in our expressions.

The second statement in our example adds $x + x$. This calls the operator+() from DivisionAlgebraCategory<V, S>; the corresponding user-must-define function from FunctionalAlgebra<Domain> that is called is operator+=():

Function/FunctionalAlgebra

```
template<class Domain>
FunctionalAlgebra<Domain>& FunctionalAlgebra<Domain>::
operator+=(const FunctionalAlgebra<Domain>& rhs) {
    f = new AddFunctional<Domain>( f, rhs.f );
    return *this;
}
```

This function creates an AddFunctional<Domain> object from the operands of the + operation and holds the sum representation with a pointer to its IsoFunctional<Domain> interface.

The AddFunctional<Domain> itself is implemented like this:

Function/BinaryFunctional.h

```
template<class Domain>
class AddFunctional :
    public virtual IsoFunctional<Domain>,
    private BinaryFunctional<Domain> {
public:
    AddFunctional(
            const CloneableObjPtr< IsoFunctional<Domain> >& lhs,
            const CloneableObjPtr< IsoFunctional<Domain> >& rhs
        ):
        BinaryFunctional<Domain>(lhs, rhs){
}

virtual Domain operator()(const Domain& v) const { return lhs()(v) + rhs()(v);}

virtual AddFunctional<Domain>*
    clone() const { return new AddFunctional<Domain>(*this); }
};
```

The constructor initializes the base class (described in the next paragraph) and the function call operator implements addition of the left-hand (lhs()) and right-hand (rhs()) sides of the base class representation.

The base class representation is simply two cloneable object pointers pointing at functional interfaces:

Function/BinaryFunctional.h

```
template<class Domain>
class BinaryFunctional {
```

```
public:
    BinaryFunctional(
            const CloneableObjPtr<IsoFunctional<Domain> >& lhs,
            const CloneableObjPtr<IsoFunctional<Domain> >& rhs
        ):
        the_lhs(lhs),
        the_rhs(rhs) {
    }
    const IsoFunctional<Domain>& lhs() const { return *the_lhs; }
    const IsoFunctional<Domain>& rhs() const { return *the_rhs; }
private:
    CloneableObjPtr< IsoFunctional<Domain> > the_lhs;
    CloneableObjPtr< IsoFunctional<Domain> > the_rhs;
};
```

Other functionlike classes for each algebraic operation are derived from BinaryFunctional<Domain>, adding the appropriate function evaluation operation.

When the expression f(5) is evaluated, the value 5.0 is passed down through the function call operator of FunctionalAlgebra<Domain> for the object f to the function call operator in AddFunctional<Domain>. This object passes 5.0 to both its left- and right-hand operands, signaling these operands to evaluate themselves at $x = 5$. In this example both operands are instances of XFunctional<Domain> representing x. These objects return the 5.0 to the AddFunctional<Domain> for both lhs() and rhs(). The results are added and returned as the value of AddFunctional<Domain>'s operator()(), and finally the FunctionalAlgebra<Domain> object returns f(5.0)=10.0.

Figure 17.2 illustrates the objects created for $f = x + x$. An AddFunctional holds two functions, the left- and right-hand operands of an addition expression in its BinaryFunctional base component. When called as a function, the AddFunctional object calls the operand functions and returns their sum. Focusing on the arrows, the object diagram branches at the AddFunctional<Domain>. More complex expressions result in more branches, and hence this structure is called an *expression tree*.

Unary operators are implemented similarly by derivation from a UnaryFunctional<Domain> base class, and constants of the algebra are implemented as a no-argument functional object that returns a stored value set in the constructor:

Function/ConstantFunctional.

```
template<class Domain>
class ConstantFunctional :    // Always returns its initial value.
    public virtual IsoFunctional<Domain> {
public:
    ConstantFunctional(Domain r) : the_obj(r){ }

    virtual Domain operator()(const Domain& v) const { return the_obj; }
```

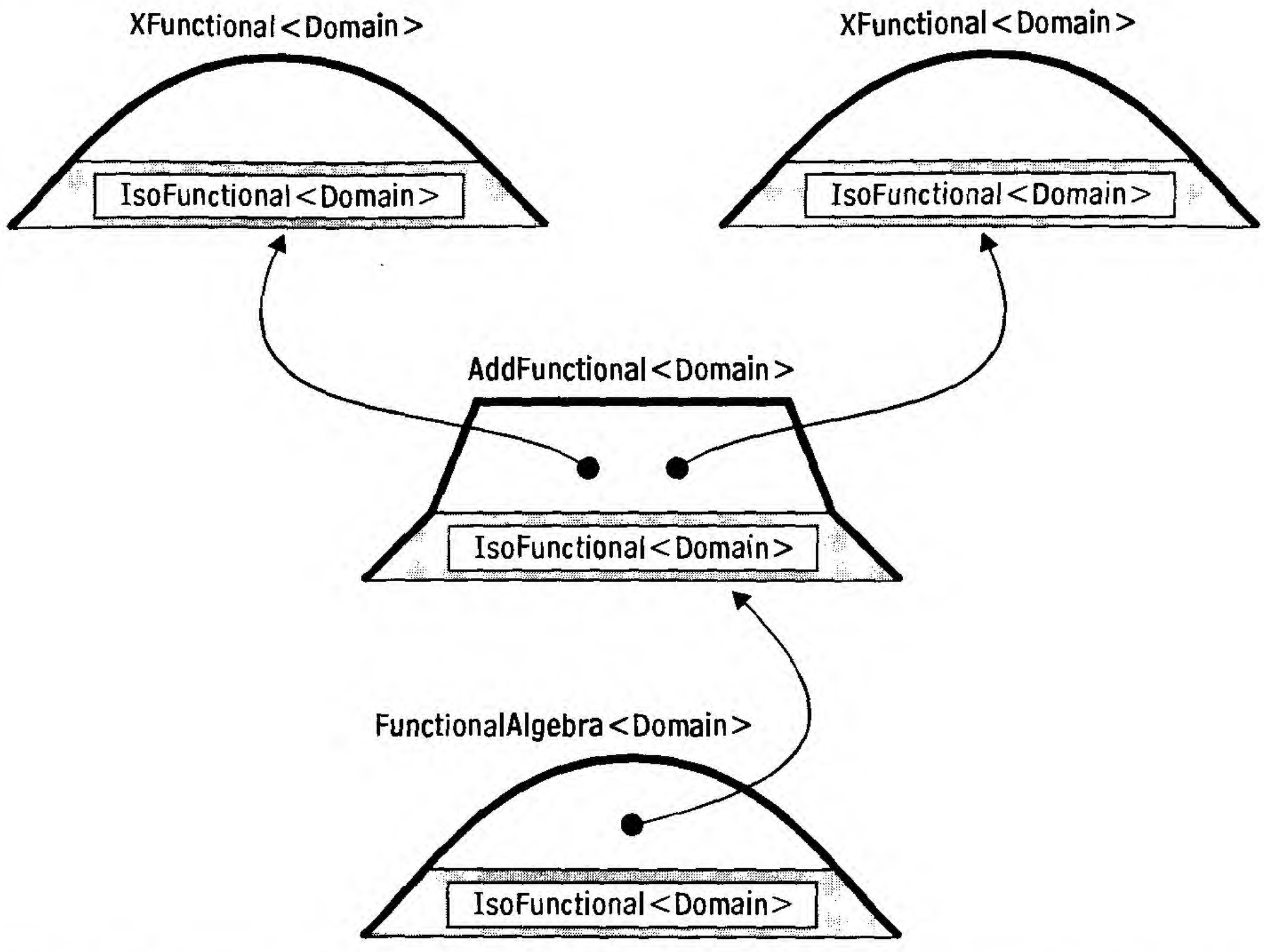


Figure 17.2 Object Sketches for a FunctionalAlgebra<Domain> Expression $f = x + x$. The object on the bottom holds a pointer to an object with an IsoFunctional<Domain> interface; here it is an AddFunctional<Domain> object that represents $x + x$. Inside this object are two pointers, to the left- and right-hand operands of the sum. In this case both point to the placeholder object x, an XFunctional<Domain> that returns its argument value on every call.

```
    virtual ConstantFunctional<Domain>* clone() const {
        return new ConstantFunctional<Domain>(*this);
    }
private:
    Domain the_obj;
};
```

Parameters are implemented by holding onto reference-counted pointers that allow independent setting of the parameter outside of the expression object:

Function/ParameterFunctional.h

```
template<class Domain>
class ParameterFunctional :
    public virtual IsoFunctional<Domain> {
```

```
public:
    ParameterFunctional(const Domain& r) : the_range_obj( new Domain(r) ) { }
    ParameterFunctional(const CountedBuiltInPtr<Domain>& r) : the_range_obj(r) { }

    virtual Domain operator()(const Domain& v) const { return *the_range_obj; }
    virtual ParameterFunctional<Domain>*
        clone() const { return new ParameterFunctional<Domain>(*this); }

    Domain&       value()       { return *the_range_obj; }
    const Domain& value() const { return *the_range_obj; }
private:
    CountedBuiltInPtr<Domain> the_range_obj;
};
```

Construction of a FunctionalAlgebra object with a counted pointer creates one of these objects:

Function/FunctionalAlgebr

```
template<class Domain>
FunctionalAlgebra<Domain>::FunctionalAlgebra(const CountedBuiltInPtr<Domain>& d) :
    f( new ParameterFunctional<Domain>(d) ) {
}
```

For example, suppose that you want to represent the sum of two Gaussian functions, $\exp((x-\mu)^2/\sigma^2)$, each having the same mean μ but different widths $\sigma = \sigma_1$ and $\sigma = \sigma_2$. You also want to be able to change the common mean at will. The following class represents the Gaussian function:

Function/GaussianFunctional

```
template<class Domain>
class GaussianFunctional :
    public virtual IsoFunctional<Domain> {
public:
    GaussianFunctional(
        const IsoFunctional<Domain>& mean,
        const IsoFunctional<Domain>& width
    ) :
    the_mean(mean),
    the_width(width) {
}

GaussianFunctional(const Domain& mean, const Domain& width);
virtual Domain operator()(const Domain& v) const {
    return exp( - sqr( (v - (*the_mean)(v)) / (*the_width)(v) ) );
}
```

```
    virtual GaussianFunctional<Domain>* clone() const {
        return new GaussianFunctional<Domain>(*this);
    }
private:
    CloneableObjPtr< IsoFunctional<Domain> > the_mean;
    CloneableObjPtr< IsoFunctional<Domain> > the_width;
};
```

We set up the function by first creating a reference-counted pointer to the common mean, creating FunctionAlgebra<T> instances for the shared mean and the two widths, creating the two Gaussian functions, and finally creating the sum of the Gaussians:

ch17/tSharedMean.C

```
CountedBuiltInPtr<double> p_mean(new double);
FunctionalAlgebra<double> mean(p_mean);
FunctionalAlgebra<double> width1(1.0);
FunctionalAlgebra<double> width2(2.0);
GaussianFunctional<double> g1(mean, width1);
GaussianFunctional<double> g2(mean, width2);
FunctionalAlgebra<double> f = g1 + g2;
```

We can now evaluate the function with different means simply by setting the common mean parameter:

ch17/tSharedMean.C

```
*p_mean = 0.0;
cout << f(.5) << endl;
*p_mean = 2.0;
cout << f(.5) << endl;
```

Thus far we've seen how we can build objects at runtime that represent functions. However, all of the functions we have built could just as well have been written as functions in C++ to be compiled and executed directly. The next section illustrates how the functional algebra system can be used to build functions from user input.

17.6 Example: A Simple Function Evaluator

The expressions represented by FunctionalAlgebra<Domain> can be built during program execution. As an example, we build a simple function evaluator that reads numbers, function names, and the letter x as a placeholder for a variable, allowing an expression like add(1.0, exp(multiply(−0.5, x))) to be entered and then evaluated for various values of x. We simplify the mechanics of analyzing the user's input by restricting ourselves to functions, rather than allowing both functions and operators, and allowing only a single (unnamed) function of a single

variable to be defined. Constructing the function will be simple, given the FunctionalAlgebra<Domain> template.

The expressions allowable for this example are defined by the following simple grammar, in much the same style encountered in manuals like the *The Annotated C++ Reference Manual* [44]. The input is partitioned into a sequence of *tokens*: a number (e.g., −0.5), a word (e.g., exp), the letter x, a comma, or a left or right parenthesis. These tokens can appear according to the following grammar:

function:
word (*termlist*)
termlist:
term
termlist , *term*
term:
number
x
function

Each italicized word is a *nonterminal*, and each of the lines appearing below a nonterminal is a *production* that defines the nonterminal. Each nonterminal represents a sequence of tokens that follow the pattern specified by any one of the productions on the lines below it in the grammar. The nonterminal *function* is the *start symbol* for this grammar (i.e., all valid inputs must match the pattern specified by *function*).

The process of matching an input string against the grammar is called *parsing* the input. For example, let's parse the input add(1.0,exp(multiply(−0.5,x))). It is a *function* with a word add and a *termlist* 1.0,exp(multiply(−.05,x)); that *termlist* has a *termlist* with a single *term* 1.0 (a number), followed by a comma, followed by a *term*, exp(multiply(−0.5,x)). That term is itself a function with word exp and so on. The entire process is illustrated in Fig. 17.3.

We divide our function analyzer into two parts, a *lexical analyzer*, a component that iterates through the tokens of an input string, and a FAMaker class that analyzes the terms and builds functions. The lexical analyzer looks like one of our usual iterators:

ch17/Lexer.h

```
class Lexer {
public:
    Lexer(const String& s);
    enum TokenKind { open, close, number, x, word, comma };
    TokenKind current() const;
    void advance();
    Boolean more() const;
```

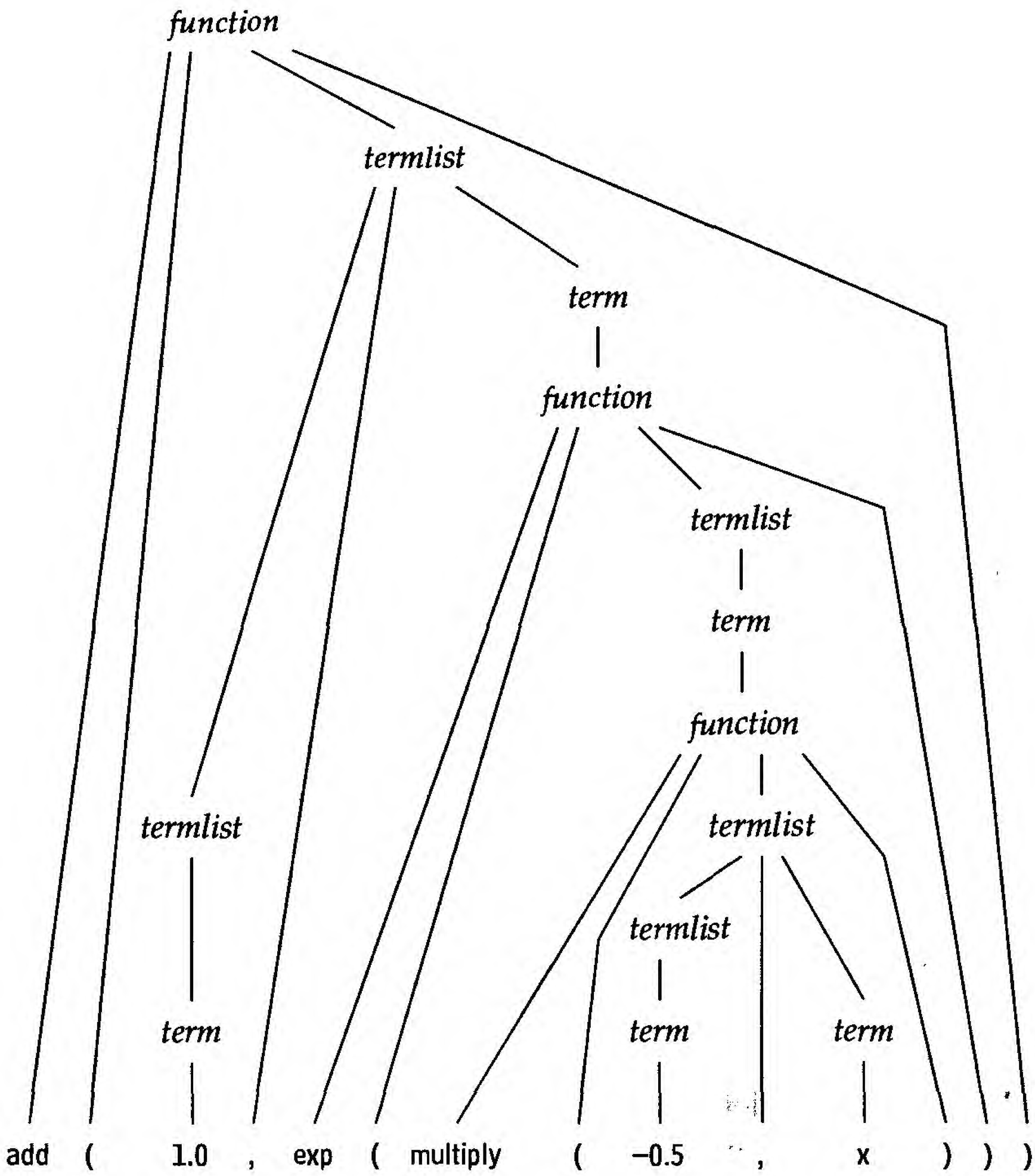


Figure 17.3 Parse of the Input add(1.0,exp(multiply(−0.5,x))).

```
    String name() const;       // Name of a word
    double value() const;      // Value of a number
    // ...
};
```

The lexer is given an input string when it is created. The current() member returns a TokenKind, indicating the kind of the current token. The more() member tests whether there is a current token, and advance() steps to the next token. If the current token is a word, its name can be obtained by calling the name() member; if it is a number, its value is obtained by calling the value() member. We leave the implementation of this class as an exercise (Exercise 17.5).

To construct a FunctionalAlgebra<double> representing the expression the user enters, we develop the class FAMaker. Its function() member will build the dynamic function from the input String passed to the FAMaker constructor:

ch17/FAMake

```
class FAMaker {
public:
    FAMaker(const String& input);
    FunctionalAlgebra<double> function();
    FunctionalAlgebra<double> term();

    FunctionalAlgebra<double> exp();
    FunctionalAlgebra<double> add();
    FunctionalAlgebra<double> subtract();
    FunctionalAlgebra<double> multiply();
    FunctionalAlgebra<double> divide();
    FunctionalAlgebra<double> cos();
    FunctionalAlgebra<double> sin();
    // Add functions here and in constructor.

    // ... Exception classes SyntaxErr and FunctionUndefined ...
private:
    Lexer lexer;

    typedef FunctionalAlgebra<double> (FAMaker::*ParserFunction)();
    ConcreteFormedArray1d<String> function_names;
    ConcreteFormedArray1d<ParserFunction> functions;
    ParserFunction lookup(const String& function_name) const;

    void skipComma();
};
```

The term() member processes a single term and returns a FunctionalAlgebra<double> instance that represents the term expression. If the term is neither a number nor x, then function() will be called to convert the word into a function. The members like exp(), add(), and so on implement the allowable functions, each returning a FunctionalAlgebra<double> instance representing the function and its arguments. Following these are some exception classes (not shown) and some private member functions and data; these are described later in this chapter as necessary.

The FAMaker constructor sets up a Lexer for the input string that will do lexical analysis and two paired arrays, one with a function name and one with a member function pointer:

ch17/FAMaker.C

```
FAMaker::FAMaker(const String& input) :
    function_names(7), // Increase size for each new function.
    functions(7),
    lexer(input) {
    function_names(0) = "exp";      functions(0) = exp;
    function_names(1) = "add";      functions(1) = add;
    function_names(2) = "subtract"; functions(2) = subtract;
    function_names(3) = "multiply"; functions(3) = multiply;
    function_names(4) = "divide";   functions(4) = divide;
    function_names(5) = "sin";      functions(5) = sin;
    function_names(6) = "cos";      functions(6) = cos;
    // Add new function name and load member function pointer here.
}
```

The name corresponds to *word* in our grammar. The paired arrays act like a dictionary in which a word can be looked up in the name array and a member function pointer to the corresponding function is returned as the definition. The private member lookup() does this work:

ch17/FAMaker.C

```
FAMaker::ParserFunction FAMaker::lookup(const String& function_name) const {
    for (Subscript i = 0; i < function_names.numElts(); i++) {
        if (function_names(i) == function_name) return functions(i);
    }
    throw FunctionUndefined(function_name);
}
```

A more flexible implementation would use a linked list like that on page 170 (Exercise 17.6).

The term() member function, which consumes and analyzes a term, has the structure of the corresponding production in the grammar:

ch17/FAMaker.C

```
FunctionalAlgebra<double> FAMaker::term() {
    if (lexer.more()) {
        if (lexer.current() == Lexer::number) {
            double val = lexer.value();
            lexer.advance();
            return new ConstantFunctional<double>(val);
        }
        else if (lexer.current() == Lexer::x) {
            lexer.advance();
            return new XFunctional<double>();
        }
```

```
        else if (lexer.current() == Lexer::word) return function();
    }
    throw SyntaxErr();
}
```

It creates and returns a FunctionalAlgebra<double> object, constructed from a ConstantFunctional<double> if a number is seen, from an XFunctional<double> if an x is seen, and from a function if a word is seen. In the latter case, the member function() is called to do the work:

ch17/FAMake

```
FunctionalAlgebra<double> FAMaker::function() {
    if (lexer.more() && lexer.current() == Lexer::word) {
        ParserFunction f = lookup(lexer.name());
        lexer.advance();
        if (lexer.more() && lexer.current() == Lexer::open) {
            lexer.advance();
            FunctionalAlgebra<double> alg = (this->*f)();
            if (lexer.current() == Lexer::close) {
                lexer.advance();
                return alg;
            }
        }
    }
    throw SyntaxErr();
}
```

The private member lookup() is called to obtain a pointer to the member function that implements the function specified by the user. Then the lexer is advanced past the opening parenthesis and the member function called through the member function pointer. Finally the lexer is advanced past the closing parenthesis and the resulting FunctionalAlgebra<double> object is returned.

Of the function-implementing member functions we show two, exp() with one argument and add() with two arguments:

ch17/FAMaker.

```
FunctionalAlgebra<double> FAMaker::exp() {
    return ::exp(term());
}

FunctionalAlgebra<double> FAMaker::add() {
    FunctionalAlgebra<double> lhs = term();
    skipComma();
    FunctionalAlgebra<double> rhs = term();
    return lhs+rhs;
}
```

These functions know both the FunctionalAlgebra<double> function that must be invoked to construct the correct function (e.g., exp()) and the number of arguments. Each argument is a *term*, which is analyzed by calling term(). In the two-argument case, the private member skipComma() is called to step the lexer past the separating comma:

ch17/FAMaker.C

```
void FAMaker::skipComma() {
    if (lexer.more() && lexer.current() == Lexer::comma) {
        lexer.advance();
        return;
    }
    throw SyntaxErr();
}
```

The analysis of the user's input proceeds from the outermost function name that the user typed in, toward more nested functions names until numbers or x objects are found. Then FunctionalAlgebra<double> objects are created as the members of FAMaker return. Once we are back at the top-level function() call, we have a dynamically created function.

Here is the main() function we used to test this class:

ch17/tFAMaker.C

```
int main(int argc, char* argv[]) {
    if (argc != 2) {
        cerr << argv[0] << ": One command line argument is expected; ";
        cerr << "it should contain an expression to evaluate" << endl;
        return 1;
    }

    FAMaker parser(argv[1]);
    FunctionalAlgebra<double> f = parser.function();
    double x;
    while (cin >> x) cout << argv[1] << "(" << x << ") = " << f(x) << endl;
    return 0;
}
```

When main() is defined with arguments, as it is here, the arguments specify pa-
■4 rameters passed to the program when it is invoked, often called *command line arguments*. The argc argument is set to the number of command line arguments, with the program name considered to be the first command line argument. Each element of the array argv is a built-in character string containing the corresponding argument; argv[0] is always the name used to invoke the program or the null string. (On many systems, a command line argument that contains blanks or special characters like parentheses must somehow be quoted. For example, on Unix systems the argument is enclosed in single quotes.)

If the correct number of arguments is given, a FAMaker is created, initialized from the expression given as a command line argument. Values of x are then read from cin, the function evaluated at x, and the result printed. For the expression add(1.0,exp(multiply(−0.5,x))), we get

```
add(1.0,exp(multiply(−0.5,x)))(−2) = 3.71828
add(1.0,exp(multiply(−0.5,x)))(1.386) = 1.50007
```

This example can be extended to more functions and more sophisticated grammars including operators. The power of function objects becomes even more evident when we consider that any arithmetic type—vectors, matrices, rational numbers, and so on—can be used with FunctionalAlgebra<Domain>. Classes following the pattern FAMaker can be written to allow functions over such various domains to be entered at runtime. The overhead of indirections and function calls makes function objects evaluate more slowly than compiled arithmetic expressions, but the latter cannot be created dynamically.

17.7 Functions over Collections

Many computations dealing with collections can be expressed naturally as the evaluation of a function on each element of the collection. For example, in modeling seismic activity, various functions representing earth responses versus time can be evaluated on the time scale of a seismic reading and compared to the measured earth motion readings. Sometimes evaluation of a function over a collection of independent values is followed by a process that accumulates the results. For example, computing the sum of the squares of the elements of a collection can be viewed as computing the square function on each element and then accumulating sums as the collection is traversed. Computing the 1-norm of a vector of numbers (i.e., the sum of the absolute values) can be viewed similarly. In this section, we explore various mechanisms for abstracting the common aspects of such computations in terms of iterators and function objects.

We begin with an iteration that assigns the value of a function applied to each element back into that element. This is the capability provided by the apply() function templates, one for interfaced arrays and one for concrete arrays:

Function/FunctionApplication.c

```
template<class T>
void apply(T (*f)(T), Array1d<T>& a) {
    for (Array1d<T>::IteratorType a_i(a); a_i.more(); a_i.advance()) {
        a_i.current() = f(a_i.current());
    }
}
```

```
template<class T, class Subscriptor>
void apply(T (*f)(T), ConcreteArray1dRef<Subscriptor, T> a) {
ConcreteArray1dRef<Subscriptor, T>::IteratorType a_i(a);
    for (; a_i.more(); a_i.advance()) {
        a_i.current() = f(a_i.current());
    }
}
```

For example, the elements of an array a can be replaced with their square roots like this:

Function/tApply.C

```
FormedArray1d<double> a(10);
for (int i = 0; i < a.shape(0); i++) a(i) = i;

apply(sqrt, a);
cout << a << endl;
```

When the transformation applied to a collection acts like an operation between consecutive elements, as in $s = a + b + c + \cdots z$, the result is a single object. This transformation can be written with browsers and member function pointers for the objects, as in the following function template:

Function/reduce.h

```
template<class Browseable>
Browseable::EltT
reduce(const Browseable& b, void(Browseable::EltT::*f)(Browseable::EltT)) {
        // Apply the member function "T::f(T)" between successive elements of "b".

    Browseable::BrowserType i(b);
    if (!i.more()) return Browseable::EltT();

    // Start with result being first element, then apply f to current
    // result and next element.
    Browseable::EltT result = i.current();
    i.advance();
    while (i.more()) {
        (result.*f)(i.current());
        i.advance();
    }
    return result;
}
```

It can be used like this:

ch17/tReduce.C

```
FormedArithmetic1d<complex> pa(40);
pa=complex(1.0,1.0);
complex sum = reduce(pa, complex::operator+=);
```

17.8 Summary

Many physical phenomena are modeled as functions; therefore many scientific and engineering programs manipulate functions. As we have illustrated here, C++ provides ways to manipulate functions that are safe and convenient. Forms similar to the ones we show here help make programs more abstract: Instead of specific C++ functions for transforming specific C++ arrays, the transformation itself is abstracted and wider ranges of functions and collections can be used.

Thinking about function objects also gives us a different point of view on the relation between functions and objects. For example, think back to the array of GPIBInstrument objects in Section 9.9. Now we can recognize that using heterogeneous objects (voltage supplies and voltmeters, for example) through an array of homogeneous interface pointers (GPIBInstrument*) is a kind of collection transformation. Passing send() to every instrument acts to transform these instruments through their common interface in a way that is similar to applying sqrt() to every member of a collection of double objects, as we did in this chapter.

17.9 Notes and Comments

17.1 Most mathematical languages provide extensive function application procedures; see, for example, Mathematica [120] and APL [45, 19].

17.2 The FunctionalAlgebra<Domain> class template started out more generally as a map from Domain to Range, rather than from Domain to Domain. However, it seemed that the scope of real problems that can be solved with the more general version did not justify its complexity. Numerical work almost always requires functions with coincident range and domain.

17.3 The examples applying functions to collections in Section 17.7 illustrate a way of abstracting the control flow of a loop from the action of the function object (function application). Other control-flow structures can be abstracted, the classic example being object-oriented menu-driven user interfaces. In such interfaces, the menu-item object both controls its visual representation on the screen and contains an action to be performed when selected. In place of the cycle of obtaining user input, classifying it, and branching to an action typical of command-driven interfaces like the FAMaker example, the user input on the menu item triggers its contained action. The menu appearance on the user screen abstracts the branching control-flow logic; by filling in different menu-item objects different possible actions become allowed.

17.4 Many LISP systems have *streams* (different from C++'s I/O streams) to support the kind of delayed evaluation we have used with browsers in Section 17.7. Instead of computing an entire collection and copying it multiple times, the elements of the collection are computed as needed. This technique often leads to concise, elegant programs with reasonable efficiency. See [1, Section 3.4] for many examples.

17.10 Exercises

17.1 Redefine NormalDensityFunction<Domain> from page 515 to put it in category IsoFunctional<Domain>.

17.2 IsoFunctional<Domain> (page 516) redeclares the clone() function inherited from Functional<Domain, Domain>. Why? *Hint:* Consider using CloneableObjPtr< IsoFunctional<Domain> > pointers.

17.3 Design, implement, and test a CachingFunction<Domain, Range> class template with a constructor taking a built-in pointer to a C++ function such that the following code:

ch17/tCachingFunction.

```
float f(float x) { // Expensive computation
    cout << "f called" << endl;
    return 2 * x;
}

int main() {
    CachingFunction<float, float> f_cache(f);
    cout << f_cache(1) << endl;
    cout << f_cache(1) << endl;
    cout << f_cache(2) << endl;
    cout << f_cache(2) << endl;
    cout << f_cache(3) << endl;
    return 0;
}
```

produces the following output:

```
f called
2
2
f called
4
4
f called
6
```

These objects cache the return value of the immediately preceding function call.

17.4 Write and demonstrate a MappingBrowser<Domain> class that transforms elements from a collection as it is iterated. The constructor should take a browser argument and an IsoFunctional<Domain> argument. Each call to the MappingBrowser<Domain> current() should transform the current() element of the contained browser. You may change the template parameters to increase the potential application.

17.5 Implement the Lexer class on page 527. Test it on a variety of inputs and discuss your decisions on error handling and the impact they had on your class and its usability.

17.6 Change the implementation of FAMaker to replace the paired arrays that implement the dictionary with a List<ParserFunctionEntry> object, where you have defined ParserFunctionEntry class to hold the parser function pointers and names. Discusss the advantages and disadvantages of this solution compared to the one in this chapter.

17.7 The *Legendre polynomials* $P_n(x)$ (see [28, Section II.8]) are given by the following recurrence relation:

$$(n+1)P_{n+1}(x) - (2n+1)xP_n(x) + nP_{n-1}(x) = 0$$

with $P_0 = 1$ and $P_1 = x$. This relation can be rewritten as

$$P_n(x) = \frac{2n-1}{n}xP_{n-1}(x) - \frac{n-1}{n}P_{n-2}(x).$$

Define function objects to represent Legendre polynomials. The constructor should take an unsigned integer giving the polynomial order. Using this class, get the following code to work:

ch17/tLegendre.C

```
LegendrePolynomial<double> p2(2);
cout << p2(10) << endl;
```

17.8 Although many mathematical functions satisfy recurrence relations, it is not always wise to evaluate the functions directly from the recurrence relations because of round-off error. However, *Clenshaw's recurrence formula* often provides a numerically stable way to evaluate functions defined in this way. See [93, Section 5.5]. Write a function template or function object class to implement Clenshaw's recurrence formula.

CHAPTER 18

Using Legacy Libraries

IMPLEMENTING classes for use in scientific and engineering applications often involves numerical algorithms or other information processing using established algorithms. Many well-tested and documented non-C++ subroutine libraries exist for such algorithms. These libraries are mostly written in FORTRAN, with some newer libraries written in C. Fortunately, C++ programs can use many of these libraries, with more or less thought and work, depending on the library's purpose, design, and implementation. In this chapter, we discuss the issues involved in using such *legacy* FORTRAN and C subroutine libraries, illustrating the ideas with practical examples.

A legacy library can be used by merely calling the FORTRAN subroutines or C functions as C++ functions. This is conceptually simple, requiring only that the mechanics of calling one language from the other be mastered. Where the mechanics are burdensome, it pays to wrap each foreign language function in a C++ function so that the interlanguage adaptation is local to one piece of code.

A more object-oriented approach uses functions from the legacy library to implement the behavior of C++ objects, an approach called *external object implementation* [35, 36]. This is the approach we took in Chapter 15. We first identified and designed the objects, including data stored in a FORTRAN-compatible form, and used LAPACK subroutines to implement the behavior of the objects. We suggest the following:

- Wrap non-C++ libraries in classes matching the design of the library.

Then use the resulting classes to implement your original system. This two-step procedure helps prevent the structure of the non-C++, non–object-oriented library from unduly influencing your overall design.

Section 18.1 discusses the mechanics of working with C functions from C++. Section 18.2 then illustrates external object implementation of a String class, using the ANSI C string library for the implementation. We then move on to FORTRAN, starting with mechanics in Section 18.3. Since so many subroutine libraries for scientific and engineering applications are written in FORTRAN, we look at several

techniques and examples of using FORTRAN libraries. Section 18.4 looks at techniques for exploiting a subroutine naming pattern commonly found in FORTRAN libraries; we use LAPACK as the example, completing the implementations of the classes designed in Section 15.4. Section 18.5 illustrates how to extend the concrete array system developed in Chapter 13 to work with packed representations of matrices with special structure, again using LAPACK as the example. Section 18.6 shows how to combine the algebraic structure categories of Chapter 16 with the concrete, FORTRAN-compatible arrays of Chapter 13 and the computational capabilities of the popular BLAS (Basic Linear Algebra Subroutines) libraries [69, 41, 42] to implement a system of matrices. We conclude, in Section 18.7, by combining all of this work into classes for solving linear systems using LAPACK's implementation of singular value decomposition. These classes form a basis for the extended example in Chapter 19.

18.1 Working with C

C++ is roughly a superset of ANSI C, so the mechanics of working with C functions in C++ are simple. In fact, many ANSI C functions are correct C++ functions with the same meaning. In such cases, the functions can be compiled with your C++ compiler and used directly as C++ code. Even when this is not possible (e.g., you don't have source for the C code) or not desirable, code compiled by a vendor's C++ compiler is generally compatible with code compiled by that same vendor's C compiler.

M §18

The remainder of this section describes the mechanics of working with C functions in C++. Section 18.2 then illustrates these mechanics using a C function library to implement a character string class.

18.1.1 Calling C Functions from C++

The C++ language definition requires that it be possible to call C functions from C++. The ordinary extern function declaration (Section 5.2.3) specifies not only external linkage but also, implicitly, that the function is a C++ function. Another language can be specified explicitly. To call a C function, the linkage type "C" follows the extern keyword:

M §7.4

ch18/miscdcls.C

```
extern "C" int rand();    // C random number generator
```

Several functions can be given linkage type "C" like this:

ch18/miscdcls.C

```
extern "C" {
    // ANSI C random number generator functions
    int rand();
    void srand(unsigned int seed);
}
```

Indeed, a C++ header file can be obtained from a C header file like this:

ch18/miscdcls.C

```
extern "C" {
#include "existlib/test/myCfuncs.h"
}
```

Many C++ compilers come with a set of header files that use this technique to provide access to all of the functions in the standard C library.

Since C++'s built-in types are identical to C's built-in types, arguments can be passed to C functions directly, with no mapping of types required. It is usually necessary to link the combined C++ and C programs using C++'s linking command, which arranges for C++ static constructors to be run; consult your compiler's manuals for details.

18.1.2 Calling C++ Functions from C

It is occasionally useful to provide a C-callable C++ function. For example, the authors of the C functions in the book *Numerical Recipes in C* chose to report errors by calling the function nrerror() [93, Section 1.0]. The nrerror() function provided prints a message and terminates the program. The book recommends modifying nrerror() to do more sophisticated error handling.

Suppose you want to call a *Numerical Recipes* function from your C++ program and you want errors to cause an exception to be thrown. You need to provide a C-callable function that throws an exception (i.e., a C++ function that can be called from C). This can be done using the extern "C" linkage specification:

ch18/nrerror.C

```
class NumericalRecipesErr :
    public SciEngErr {
public:
    NumericalRecipesErr(const String& msg);
    virtual String message() const;
private:
    String the_msg;
};

extern "C" void nrerror(char error_text[]) {
    throw NumericalRecipesErr(error_text);
}
```

This done, we can call a function from *Numerical Recipes* and handle errors as appropriate:

ch18/nrerror.

```
try {
    nrfunction();    // Function in Numerical Recipes
    // ...
} catch (const SciEngErr& e) {
    cout << e.message() << endl;
}
```

18.1.3 Input-Output

Most C programs do I/O using the functions in the standard I/O library, declared in the header file stdio.h. C++ programs can also use the functions in stdio.h, but they are not type safe and provide no means for reading and writing programmer-defined objects. (See also Notes and Comments 18.1.) Therefore we recommend the following:

- Use the iostream.h functions in C++ programs even when calling C functions that do I/O.

When C functions that perform I/O are used with C++ programs that perform I/O using iostream, you must arrange to coordinate the two. For example, if both C and C++ functions write to standard output, the characters written must appear in the order in which they are written, not intermixed. The necessary coordination can be provided by executing the following C++ statement before any I/O:

ch18/syncIO.

```
#include <iostream.h>
//...
ios::sync_with_stdio();
```

18.2 Example: String Class

To illustrate the use of C libraries for external object implementation, we build a character string type, a C++ class wrapped around an existing C subroutine library. More powerful and more efficient character string types are possible. We adopted the wrapping approach because it yields a practically useful type with a simple implementation.

18.2.1 ANSI C String Functions

ANSI C provides a collection of functions for manipulating null-terminated character strings (cf. Section 2.14). These functions are powerful, but using them correctly requires attention to many details, the most odious being memory management. For example, concatenating strings requires the programmer to create a large enough array of characters to hold the result. Here is a subroutine that takes two strings and concatenates them with an intervening blank:

ch18/blank-concat.C

```
char* blank_concat(const char* word1, const char* word2) {
    unsigned result_len = strlen(word1) + strlen(word2) + 2;
    char* result = new char[result_len];
    return strcat( strcat( strcpy(result, word1), " " ), word2 );
}
```

The first statement computes the length of the result string, including the intervening blank and the terminating null character, and the second statement allocates the necessary memory. With the memory allocated, the third statement uses the string library functions to copy in the first word, concatenate the intervening blank, and, finally, concatenate the second word. Each library function returns a pointer to its first argument, the destination string, allowing function calls to be cascaded as we have done. The calling function is responsible for deleting the result string:

ch18/blank-concat.C

```
char* p = blank_concat("Hi", "there");
// ...
delete [ ] p;
```

We'd like our wrapper class to handle the details for us.

Documentation for the ANSI C string functions is widely available (see Notes and Comments 18.4), so we provide only a brief description of the functions we use. Most of the functions are declared in the standard header string.h; a few additional functions for converting strings to numbers are declared in the standard header stdlib.h.

18.2.2 Using the String Class

We begin with a user's view of the String class. In addition to construction from built-in strings and characters, we provide square brackets indexing for compatibility with built-in character strings and function call indexing for compatibility with array subscripting; the two notations are equivalent for strings. The correspondence between the C functions and the facilities provided by our String class is given in Table 18.1.

String concatenation is denoted by addition, with the binary operator provided by a global function so that built-in character string arguments are converted (cf. Section 6.5):

ch18/demoString.C

```
String s1("abcd");
s1 += "efg";                  // Right operand converted; result: abcdefg
String s2 = "xyz" + s1;       // Left operand converted; result: xyzabcdefg
String s3 = s2 + "123";       // Right operand converted; result: xyzabcdefg123
```

Comparisons are also done by operators rather than function calls.

Purpose	*ANSI C*	String
Concatenate	strcat	+, +=
Compare	strcmp	<, <=, ==, !=, >=, >
Copy	strcpy	constructors, =, subString(Subscript)
Copy at most *n* characters of a string	strncpy	String(const char*, Subscript)
Index of first occurrence of a character	strchr	find(char)
Index of last occurrence of a character	strrchr	findlast(char)
Index of a substring in a string	strstr	find(const String&)
Index of last character from set given	strspn	span(const String& set)
Index of last character not in set given	strcspn	cspan(const String& set)
Index of first character from a set given	strpbrk	brk(const String& set)
Length	strlen	strlen()
Convert to double	strtod	strtod()
Convert to long int	strtol	strtol()
Convert to unsigned long int	strtoul	strtoul()

Table 18.1 Mapping of ANSI C String Functions to Our String Class Member Functions. Functions listed in the top portion are declared in header file string.h; the last three functions are declared in stdlib.h.

We have also departed significantly from the interface provided by the ANSI C searching functions. Typically the ANSI functions return a pointer to the beginning of the result of the search, with a null pointer indicating failure. Since we want to work with strings as objects, moving away from pointer manipulation, we decided to return search results as string indices and to use the Fallible<T> class template (page 308) to indicate success or failure. For example, suppose we have a string called filename that contains the name of a file, possibly including an extension following a period, like this:

SciEng/tString

```
String filename("/tmp/t.C");
```

We could then create a new file name with the extension out like this:

SciEng/tString

```
Subscript dot = filename.findlast('.').elseDefaultTo(filename.strlen());
String new_filename = filename.subString(0, dot) + ".out";
```

where dot gets the index of the last period in the filename or, if there is no period, the length of the file name. The second statement uses dot to select the prefix part of the file name to use in constructing the new file name.

18.2.3 Class Definition

The primitive behavior of our strings is provided by the constructors, destructor, assignment operators, indexing functions, and concatenation operators:

SciEng/String.h

```
class String {
public:
        // Constructors
    String();                               // Null string
    String(const char*);                    // From built-in string
    String(const char*, Subscript n);       // ... for at most n characters
    String(char);                           // From single character
    String(const String&);                  // From another String

        // Assignment
    String& operator=(const String&);       // Assign from String
    String& operator=(const char*);         // ... from built-in string
    String& operator=(char);                // ... from single character

        // Destructor
   ~String();

        // Indexing
    char   operator[](Subscript i) const;
    char& operator[](Subscript i);
    char   operator()(Subscript i) const;
    char& operator()(Subscript i);

        // Concatenation
    String& operator+=(const String& rhs);
    friend String operator+(const String& lhs, const String& rhs);
```

We declare the comparison operators as global functions:

SciEng/String.h

```
        // Comparisons
    friend Boolean operator==(const String&, const String&);
    friend Boolean operator!=(const String&, const String&);
    friend Boolean operator< (const String&, const String&);
    friend Boolean operator<=(const String&, const String&);
    friend Boolean operator> (const String&, const String&);
    friend Boolean operator>=(const String&, const String&);
```

Searching capabilities are provided by the following members:

SciEng/String.h

```
    // Searches
Fallible<Subscript> find   (const String& s) const;    // First occurence of s
Fallible<Subscript> find(char c)             const;    // First occurence of c
Fallible<Subscript> findlast(char c)         const;    // Last occurence of c

Fallible<Subscript>
                    brk    (const String& s) const;    // First occurence of any character in s
Subscript span  (const String& s) const;    // Size of initial span of characters in s
Subscript cspan(const String& s) const;    // Size of initial span of characters not in s
```

Various miscellaneous, conversion, and I/O functions round out the public members:

SciEng/String.h

```
    // Sub-strings
String subString(Subscript start              ) const;    // From start
String subString(Subscript start, Subscript n) const;    // ... for n characters

    // Miscellaneous
Subscript    strlen() const;    // Length of string (not including null)
const char* c_str()  const;    // Pointer to underlying built-in string
char*       c_str();           // Pointer to underlying built-in string

    // Conversion into numbers
Fallible<double>        strtod()  const;    // Conversion to double
Fallible<long>          strtol()  const;    // Conversion to long
Fallible<unsigned long> strtoul() const;    // Conversion to unsigned long

    // I/O
friend ostream& operator<<(ostream& s, const String& cs);
friend istream& operator>>(istream& s, String& cs);
```

For simplicity, the subString() members return copies of the specified substrings, not references to them. See Exercise 18.3. The c_str() members are described in Section 18.2.4.

Finally we complete the String class definition with browser and iterator nested classes, the member datum that points to the built-in character string, and a private constructor:

SciEng/String.h

```
    // Iterators
class Iterator;    // Forward declaration

class Browser {
public:
    Browser(const String& s);

    Boolean more() const;
    void    advance();
    char    current() const;

    Browser(const Iterator&);
private:
    const char* cur;
    const char* endp;
};

class Iterator {
public:
    Iterator(String& s);
        Boolean more() const;
        void    advance();
        char&   current() const;
    private:
        friend Browser::Browser(const Iterator&);
        char* cur;
        const char* endp;
    };
private:
    char* data;    // Pointer to built-in character string
    String(const String&, const String&);
};
```

The browser and iterator classes provide read-only and read-write iteration over the characters in a String, in the style of the array iterators of Section 13.7. Note that an iterator can be converted to a browser, allowing a function expecting a browser to accept an iterator. The role of the private constructor is discussed on page 550.

Our implementation of String uses a built-in pointer to an array of characters and manages storage for it. We chose not to use our concrete arrays for this purpose because the null-termination model of C strings makes the size stored in our

arrays redundant and the implementation inheritance small. In addition, String is used in the error reporting for our arrays, and coupling these two fundamental classes even further makes changing the system of classes more difficult while yielding little gain.

18.2.4 String's Exposed Implementation

The c_str() members expose the underlying representation of strings as a pointer to a null-terminated built-in character string. Ordinarily one doesn't expose a class's representation; here, however, we are explicitly wrapping built-in character strings, and we'll probably want to call various C and C++ functions that manipulate built-in strings. Although we aren't concerned in this case with violating the information hiding principle, danger nevertheless lurks whenever a class provides a pointer to its internal representation.

To see the danger, let's consider the c_str() members. They each return a pointer (const or not depending on the const-ness of the String) to the internal representation of a String. If we use this pointer after the String object is destroyed, we have a dangling reference (cf. Section 7.6). So what: We already know to avoid dangling references. The new issue here is the problem's subtlety. Consider the following code:

ch18/tempdel.C

```
extern void f(const char*);

String s1 = "abc";
String s2 = "def";
f( (s1 + s2).c_str() );
```

The expression s1 + s2 yields a temporary String object initialized to "abcdef", and
RM §12.2 c_str() returns a pointer to the object's internal representation. C++ specifies that the temporary can be destroyed immediately after its value is used—here, immediately after the call to c_str(). If the temporary is destroyed before f() is called, the pointer passed to f() would be a dangling reference because the internal representation of the temporary is deleted when the temporary is destroyed. This code can be written safely like this:

ch18/tempdel.C

```
String temp(s1 + s2);
f(temp.c_str());
```

Written this way, the temporary object is bound to a variable and therefore can't be destroyed until that variable goes out of scope:

- Where possible, a wrapper class should not provide a pointer to its internal representation.

See also Notes and Comments 18.6. When following this advice is not practical, as with the String class, substitute the following:

- When using a wrapper class, only obtain a pointer to the internal representation of an object that is bound to a variable.

We made one other design decision related to this dangling reference problem: You get a pointer to the underlying built-in character string by explicitly calling c_str(), not implicitly through a conversion to char* operator. Providing such a conversion operator would have made a subtle problem even more subtle and difficult to track down.

- Avoid conversion operators that return pointers to the internal representation of a wrapper class.

18.2.5 Function Definitions

Construction. The String constructors all initialize data to point to a newly allocated array of characters (a built-in character string), taking care that the string is null terminated. Here are two examples:

SciEng/String.C

```
String::String(const char* s) :
    data( strcpy(new char[::strlen(s) + 1], s) ) {
}

String::String(char c) :
    data(new char[2]) {
    data[0] = c;
    data[1] = '\0';    // Null terminate
}
```

The scope operator (::) is used to specify that we are calling the global strlen() function, not String's member function of the same name. This technique can be used to resolve name conflicts between wrapper class members and global functions in the library being wrapped.

Also note that the C library functions for copying and concatenating strings return pointers to the destination strings. Here the destination string is the newly allocated storage. String controls the lifetime of this storage.

Assignment. The assignment operators check for self-assignment, delete the string holding the current value of the left-hand argument, determine the length of their right-hand argument, and use the C library's strcpy() string copy function to copy in the new value. Here is one of the assignment operators:

SciEng/String.C

```
String& String::operator=(const String& rhs) {
    if (data != rhs.data) {
        delete [ ] data;
        data = strcpy(new char[rhs.strlen() + 1], rhs.data);
    }
    return *this;
}
```

Concatenation. There are two string concatenation operators, the asymmetric += operator and the symmetric + operator. When we have such operator pairs, we usually implement the symmetric operator by calling the asymmetric operator, as described in Section 6.5. Let's look at the asymmetric operator definition first:

SciEng/String.C

```
String& String::operator+=(const String& rhs) {
    char* new_data =
        strcat(strcpy(new char[this->strlen() + rhs.strlen() + 1], data), rhs.data);
    delete [ ] data;
    data = new_data;
    return *this;
}
```

The first statement allocates the memory necessary to hold the result string and copies the argument strings into the result. Then the original string is deleted and the left-hand argument's pointer is set to point to the result string.

Suppose that we follow the usual approach for implementing the symmetric operator. We would copy the left-hand argument into a temporary String and then call the asymmetric operator to do the concatenation. This is wasteful because the asymmetric operator would immediately copy the left-hand argument again. We would do better to construct the result string directly from both arguments, like this:

SciEng/String.C

```
String::String(const String& s1, const String& s2) :
    data( strcat(strcpy(new char[s1.strlen() + s2.strlen() + 1], s1.data), s2.data) ) {
}

String operator+(const String& lhs, const String& rhs) {
    return String(lhs, rhs);
}
```

The constructor takes two String objects, allocates the memory needed to hold the concatenated string, then copies the first argument into the allocated string, and finally concatenates the second argument string to the result. The symmetric operator simply calls this constructor to concatenate its operands. A constructor

used to implement an operator like this is sometimes called an *operator constructor*: It exists to implement an operator.

Comparison. All of the comparison operators are implemented using the strcmp character string comparison function, which returns either −1, 0, or 1 depending on whether its first argument is less than, equal to, or greater than its second argument:

SciEng/String.C

```
Boolean operator==(const String& lhs, const String& rhs) {
    return strcmp(lhs.data, rhs.data) == 0;
}
Boolean operator<(const String& lhs, const String& rhs) {
    return strcmp(lhs.data, rhs.data) < 0;
}
// ...
```

Searching. The string searches are implemented using the corresponding C library functions. However, most of the C search functions return a pointer to the first character of the result string, with a null pointer indicating failure. String, on the other hand, returns the subscript of the first character of the result string, with failure indicated by returning a Fallible<Subscript> object in the invalid state. Here is the code for one of the search functions:

SciEng/String.C

```
Fallible<Subscript> String::find(char find_me) const {
    const char* foundp = strchr(data, find_me);
    if (foundp)  return Fallible<Subscript>(foundp - data);
    else         return Fallible<Subscript>();
}
```

By this means, we simultaneously wrap the functionality of the C string and provide an error-handling mechanism.

Conversion to Numbers. The functions for converting a String to a number illustrate the approach to error handling adopted by the ANSI C library. Here is the function for conversion to double:

SciEng/String.C

```
Fallible<double> String::strtod() const {
    errno = 0;    // Clear error indication
    char* endp;
    double result = ::strtod(data, &endp);
    if (errno || *endp) return Fallible<double>();
    else                return Fallible<double>(result);
}
```

The variable errno is an integer declared in the standard header errno.h. C library functions set it to a positive integer when they detect an error; library functions never set it to zero.

- Check for errors detected by C library functions by clearing errno before calling the library function and checking for a nonzero value after the call.

The strtod() library function returns a number and, through its second argument, a pointer to the character after the last character it processed. For example, if given the string "123.4abc", strtod() would set the pointer to point at the a. We consider it an error if either the strtod() library function detects an error or if the entire string is not processed. When the entire string is processed, the pointer set by strtod() points at the null terminator. Thus we return an error object if either errno is non-zero or the pointer points to a non-zero value.

Input and Output. String input and output are provided by the usual << and >> operators:

SciEng/String.C

```
ostream& operator<<(ostream& s, const String& cs) {
    return s << cs.data;
}
```

SciEng/String.C

```
istream& operator>>(istream& s, String& cs) {
    const int bufsize = 100;
    char buf[bufsize];
    cs = String();
    while ( s >> setw(bufsize) >> buf ) {
        cs += buf;
        int next_input_char = s.peek();
        if (isspace(next_input_char)) {
            // Read terminated by white space, not lack of room --> stop reading
            break;
        }
    }
    return s;
}
```

Output is trivial, but input requires care because the size of the input is not known *a priori*. The input string is read into the array buf, in hundred-character chunks, by using the setw() manipulator to control maximum number of characters read. (The maximum number of characters read is one less than the number specified in

setw()'s argument, leaving room for the terminating null character.) The contents of buf are concatenated to the destination String and a test is made to determine whether the read was terminated by the size limit or by whitespace.

An input operator that works correctly for an input string of any length is another advantage provided by the wrapper class.

- Use wrapper classes to eliminate fixed maximum sizes for variable sized objects.

18.3 Working with FORTRAN

Using a FORTRAN library from C++ requires more work than using a C library because of differences in built-in types and array layout. We cover the mechanics in this section and then illustrate wrapping techniques in Sections 18.4 through 18.7, with examples drawn from calling the BLAS and LAPACK.

18.3.1 Calling FORTRAN from C++

Calling a FORTRAN subroutine from C++ requires using a linkage declaration. The C++ language definition allows, but does not require, support for "FORTRAN" linkage declarations. If your C++ compiler does not support FORTRAN linkage, you probably can call FORTRAN subroutines by declaring them with "C" linkage. See Notes and Comments 18.3. We shall proceed assuming support for "FORTRAN" linkage declarations, like this: ARM

ch18/miscdcls.C

```
extern "FORTRAN" double drand();    // Fortran random number generator
```

Although C++ has analogs of all FORTRAN types, the correspondence between C++ and FORTRAN types is not standardized. Our examples will assume the typical correspondence shown in Table 18.2. FORTRAN arrays are stored in column-major order (leftmost subscripts vary fastest), whereas C++ arrays are stored in row-major order (rightmost subscripts vary fastest). Thus FORTRAN treats a two-dimensional C++ matrix as the transpose of the matrix and vice versa. FORTRAN expects all subroutine arguments to be passed by reference, whereas C++ passes non-array arguments by value.

Such differences are accomodated easily by appropriate declaration of the FORTRAN functions. We illustrate FORTRAN calls with a few subroutines from LAPACK (cf. Section 15.1). The relevant subroutine names are shown in Table 15.2 (page 455), and their calling sequences are shown in Table 15.3 (page 456). Based on the type correspondences of Table 18.2, we declare the FORTRAN subroutines like this:

C++ Type	*FORTRAN Type*
char	CHARACTER
short int	INTEGER*2
int	INTEGER, INTEGER*4
long int	INTEGER*4
float	REAL, REAL*4
double	DOUBLE PRECISION, REAL*8
char[n]	CHARACTER*n
Class with 2 float members	COMPLEX
Class with 2 double members	DOUBLE COMPLEX, COMPLEX*16

Table 18.2 Typical Correspondence of C++ and FORTRAN Types

LapackWrap/LapackSubroutines.h

```
extern "FORTRAN" {
    // General matrices
    void sgetrf(const int& M, const int& N, float A[ ], const int& LDA, int IPIV[ ], int& INFO);
    void dgetrf(const int& M, const int& N, double A[ ], const int& LDA,
                int IPIV[ ], int& INFO);
    void sgetrs(const char TRANS[ ], const int& N, const int& NRHS, float A[ ], const int& LDA,
                const int IPIV[ ], float B[ ], const int& LDB, int& INFO);
    void dgetrs(const char TRANS[ ], const int& N, const int& NRHS,
                double A[ ], const int& LDA,
                const int IPIV[ ], double B[ ], const int& LDB, int& INFO);

    // Symmetric, positive-definite, packed matrices
    void spptrf(const char UPLO[ ], const int& N, float AP[ ], int& INFO);
    void dpptrf(const char UPLO[ ], const int& N, double AP[ ], int& INFO);
    void spptrs(const char UPLO[ ], const int& N, const int& NRHS, const float AP[ ],
                float B[ ], const int& LDB, int& INFO);
    void dpptrs(const char UPLO[ ], const int& N, const int& NRHS, const double AP[ ],
                double B[ ], const int& LDB, int& INFO);
}
```

All nonarray formal arguments are declared to be passed by reference or const reference to match FORTRAN's subroutine calling convention; array arguments are declared as arrays but are actually passed by reference because C++ converts a use of an array name to a pointer to the array's first element (cf. Section 2.11). Arguments not modified by the FORTRAN subroutine are declared const.

Declaring a function to have "FORTRAN" linkage does not affect the order in which array elements are stored: A C++ program must ensure that arrays passed to a FORTRAN subroutine are stored in the order that the FORTRAN subroutine

expects. (Of course, this is only necessary for two- and higher-dimensional arrays.) Many FORTRAN matrix processing subroutines can be told to work with matrix transposes. Alternatively, we can use C++ classes, like `ConcreteFortranArray2d<T>` (page 378), that manipulate matrices stored with the FORTRAN storage layout.

18.3.2 Calling C++ Functions from FORTRAN

Linkage declarations can also be used to declare that C++ code should be callable from FORTRAN, and this is useful when using both BLAS and LAPACK. When an illegal argument is passed to a FORTRAN subroutine in LAPACK or (most implementations of) BLAS, the FORTRAN subroutine `XERBLA` is called to handle the error. Typically this function prints an error message and issues `STOP`. For small programs with few calls to LAPACK or BLAS and with no requirement for continuous running, such error reports are adequate. For larger programs, determining the erroneous call may be difficult and stopping when runtime errors are detected may be unacceptable.

To avoid both problems, a C++ version of `XERBLA` can be defined to throw an exception:

LapackWrap/BlasSubroutines.C

```
extern "FORTRAN"
void xerbla( const char subr_name[6], int& arg_num){
    throw BlasErr(String(subr_name, 6), arg_num);
}
```

The exception class holds a copy of the subroutine name and argument number:

LapackWrap/BlasSubroutines.h

```
class BlasErr :
    public SciEngErr {
public:
    BlasErr(String subr_name, int arg_num):
        the_subr_name(subr_name),
        the_arg_num(arg_num) {
    }
    String message() const;
private:
    String the_subr_name;
    int the_arg_num;
};
```

This exception can print the error message, as the original FORTRAN version does:

LapackWrap/BlasSubroutines.C

```
String BlasErr::message() const {
    String s =
            "XERBLA from BLAS/LAPACK reports that " +
            the_subr_name +
            " has an illegal value in argument number ";
    return s << the_arg_num;
}
```

Or we can catch and handle these errors at runtime:

ch18/tXERBLA.C

```
float x[10];
// ... set x ...
try {
    float norm = Blas1Subroutines::xnrm2(10, x, 0);    // Bad third argument
} catch(const BlasErr& e) {
    userErrMessage(e);
    cerr << "Test for xerbla trap succeeds" << endl;
}
```

The exception mechanism provides an opportunity for active functions to perform cleanup operations as the destructors for automatic objects between the throw and the catch are run.

18.3.3 Input-Output

There is no general mechanism for coordinating input or output done from C++ and from FORTRAN, although there may be implementation-specific mechanisms. Thus we recommend the following:

- When mixing FORTRAN and C++ code, avoid doing input or output from both languages with the same file or device.

For example, if both a FORTRAN subroutine and a C++ function write to the display, there is a good chance that the output from the two will be intermingled on the display.

Well-written subroutine libraries separate computation and input and output, reducing the likelihood that coordinating input and output between C++ and FORTRAN will be a problem in practice. See Section 2.4 for suggestions on reading and writing FORTRAN-compatible formatted input and output with C++ streams.

18.4 Exploiting Name Commonality in Wrapped FORTRAN

In this section, we look at how the LAPACK factoring and solving subroutines can be called to implement factoring and solving for the RectLURep<T> class of Section 15.4 (page 466).

RectLURep<T> is parameterized by element type—float or double—and we need to call the corresponding LAPACK subroutine for each value of T. We could write a template specialization for each value of T that calls the appropriate LAPACK subroutine—sgetrf() or dgetrf()—but that would mean replicating much of the code by hand. Instead we introduce an additional level of function that provides name commonality that can be exploited by the RectLURep<T> template.

In other words, the LAPACK subroutine naming convention (see Table 15.1 on page 453) encodes argument types in the first character of each subroutine name; we want to remove that encoding from the name—to get name commonality—and replace it with C++ function overloading. We accomplish this by wrapping the LAPACK subroutines declared on page 554 in overloaded functions and using argument matching to select the proper function.

The overloaded function names are declared as static members of the LapackSubroutines class, primarily to aid the reader of code that calls these functions:

LapackWrap/LapackSubroutines.h

```
class LapackSubroutines {
public:
    // Factoring general matrices
    static void xgetrf(
        const int& M, const int& N, float* A, const int& LDA,
        int* IPIV, int& INFO
    );
    static void xgetrf(
        const int& M, const int& N, double* A, const int& LDA,
        int* IPIV, int& INFO
    );

    //Solving general factored matrices
    static void xgetrs(const char TRANS[],
        const int& N, const int& NRHS, float* A, const int& LDA,
        const int IPIV[],
        float* B, const int& LDB,
        int& INFO
    );
```

```
    static void xgetrs(const char TRANS[ ],
        const int& N, const int& NRHS, double* A, const int& LDA,
        const int* IPIV, double* B, const int& LDB,
        int& INFO
    );

    // Factoring symmetric, positive-definite, packed matrices
    static void xpptrf(const char UPLO[ ], const int& N, float* AP, int& INFO);
    static void xpptrf(const char UPLO[ ], const int& N, double* AP, int& INFO);

    // Solving factored symmetric, positive-definite, packed matrices
    static void xpptrs(const char UPLO[ ],
        const int& N, const int& NRHS, const float AP[ ],
        float B[ ], const int& LDB,
        int& INFO
    );

    static void xpptrs(const char* UPLO,
        const int& N, const int& NRHS, const double AP[ ],
        double B[ ], const int& LDB,
        int& INFO
    );
};
```

For example, we use the name xgetrf with floating point arguments for SGETRF and with double arguments for DGETRF:

LapackWrap/LapackSubroutines.h

```
// Factoring general matrices
inline
void LapackSubroutines::
xgetrf(const int& M, const int& N, float* A, const int& LDA, int* IPIV, int& INFO) {
    sgetrf(M, N, A, LDA, IPIV, INFO);
}

inline
void LapackSubroutines::
xgetrf(const int& M, const int& N, double* A, const int& LDA, int* IPIV, int& INFO) {
    dgetrf(M, N, A, LDA, IPIV, INFO);
}
```

A matrix may be factored to an LU representation by the following function:

LapackWrap/RectLURep.c

```
template<class T>
RectLURep<T>::Factored::Factored(RectLURep<T>::Unfactored* mp) :
    pivots( min(mp->shape(0), mp->shape(1)) ),
    facmat_p(mp)                                    {

    int info;    // info return from LAPACK factor routine.
    LapackSubroutines::xgetrf(mp->shape(0),
                              mp->firstDatum(), mp->shape(0), pivots.firstDatum(), info);
    if (info != 0) throw LapackErr::UnableToFactor(info);
}
```

The argument mp is a pointer to a ConcreteFortranArray2d<T>, representing raw FORTRAN-compatible array storage. The constructor creates a pivots array, with size based on the size of the unfactored rectangular array. The proper xgetrf function is selected and called depending on the template argument T. The solve() function is implemented similarly:

LapackWrap/RectLURep.c

```
template<class T>
void RectLURep<T>::Factored::solve(ConcreteFortranArray2d<T>& b) {
    int info;    // info return from LAPACK solve routine.
    LapackSubroutines::xgetrs(
        "No transpose", facmat_p->shape(0), b.shape(1), facmat_p->firstDatum(),
        facmat_p->shape(0), pivots.firstDatum(), b.firstDatum(), b.shape(0), info
                                );
    if (info != 0) throw LapackErr::UnableToSolve(info);
}
```

Since many FORTRAN subroutine libraries have adopted the BLAS convention for encoding element types in the subroutine name, the wrapping technique we have illustrated in this section is widely applicable.

- Use wrapping to replace element-type encoding in FORTRAN subroutine names with C++ function overloading based on argument types.

18.5 Packed Array Representations

Many FORTRAN subroutine libraries for matrices, including the BLAS [69, 41, 42] and LAPACK [5], define storage formats for matrices with special properties. These can also be represented by classes that fit within our system of array classes.

Continuing with the LAPACK example of Section 15.4, the SymPosDefPackedLU-Rep<T> class (defined on page 467) represents symmetric arrays stored in LAPACK's packed storage format using ConcreteFortranSymmetricPackedArray2d<T>: packed columns in a one-dimensional array. In this format, the symmetric array

$$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{12} & a_{22} & a_{23} & a_{24} \\ a_{13} & a_{23} & a_{33} & a_{34} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{matrix}$$

would be stored in a 10-element, one-dimensional array like this (upper triangle):

$$a_{11}\,\underbrace{a_{12}a_{22}}\,\underbrace{a_{13}a_{23}a_{33}}\,\underbrace{a_{14}a_{24}a_{34}a_{44}}\,.$$

This storage layout and many other packed array representations used in FORTRAN libraries meet the requirements necessary to fit into our concrete array system: The elements are stored in contiguous memory and the address of an element can be computed as a function of subscripts and array shape information.

To write ConcreteFortranSymmetricPackedArray2d<T>, we simply derive from ConcreteArray2d<Subscriptor, T>:

Array/ConcreteFortranSymmetricPackedArray2d.h

```
template<class T>
class ConcreteFortranSymmetricPackedArray2d :
    public ConcreteArray2d<FortranSymmetricPackedSubscriptor, T> {
public:
    ConcreteFortranSymmetricPackedArray2d(Subscript s0, Subscript s1);
    ConcreteFortranSymmetricPackedArray2d(
        const ConcreteFortranSymmetricPackedArray2d<T>&);

    ~ConcreteFortranSymmetricPackedArray2d();

    ConcreteFortranSymmetricPackedArray2d<T>&
        operator=(const ConcreteFortranSymmetricPackedArray2d<T>& rhs);
    ConcreteFortranSymmetricPackedArray2d<T>& operator=(const T& rhs);

    void reshape(const SubscriptArray<2>& s) { reshapeOnHeap(s); }
};
```

Defining the Subscriptor is the interesting part.

In our concrete array system, a Subscriptor must supply basic array shape information, an offset() function that maps indices to elements, and a projection subscriptor:

Array/ConcreteFortranSymmetricPackedArray2d.h

```
class FortranSymmetricPackedSubscriptor {
public:
    FortranSymmetricPackedSubscriptor(const SubscriptArray<2>& s);
    Dimension       dim()               const { return 2;              }
    Subscript       shape(Dimension)    const { return n;              }
    Subscript       numElts()           const { return (n * (n+1)) / 2; }
    void            setShape(const SubscriptArray<2>& s);

    Subscript offset(const SubscriptArray<2>& s) const;

    class ProjectionT {
    public:
        ProjectionT(const FortranSymmetricPackedSubscriptor&, Dimension, Subscript);
        Dimension dim()               const { return 1; }
        Subscript  shape(Dimension)   const { return n; }
        Subscript  numElts()          const { return n; }
        Subscript offset(const SubscriptArray<1>& s) const;
    private:
        const Subscript proj_sub;     // Subscript in projected-out dimension
        const Dimension d;            // Projected-out dimension
        const Subscript n;            // Number of elements in projection
    };

    ProjectionT projectionSubscriptor(Dimension d, Subscript s) const;
protected:
    Subscript n;
};
```

The offset computation depends on whether the selected element lies above the diagonal and exploits the symmetry of the array:

Array/ConcreteFortranSymmetricPackedArray2d.h

```
inline
Subscript
FortranSymmetricPackedSubscriptor::offset(const SubscriptArray<2>& s) const {
        Subscript i = s(0);
        Subscript j = s(1);
        return i <= j ? i+(j*(j+1))/2 : j+(i*(i+1))/2;
}
```

Note that the offset expressions must be written carefully to avoid dividing an odd integer by two.

We've written the projection subscriptor as a nested class within the array subscriptor. Its offset computation is different from the offset computations for the

strided layout projections we saw in Section 13.4.3: Computing the offset requires knowing both the subscript in the projected-out dimension and the subscript in the projection. (The projections for the strided arrays did not need to know the subscript in the projected-out dimension because that information is encoded by setting the data pointer to the first element in the projection. This is not sufficient with packed arrays because the offset computation requires knowing whether the selected element is below the diagonal.) Thus the projectionSubscriptor() function passes the subscript in the projected-out dimension to the projection subscriptor's constructor:

Array/ConcreteFortranSymmetricPackedArray2d.h

```
inline
FortranSymmetricPackedSubscriptor::ProjectionT
FortranSymmetricPackedSubscriptor::
projectionSubscriptor(Dimension d, Subscript s) const {
    return ProjectionT(*this, d, s);
}
```

The constructor saves this information in the projection subscriptor object

Array/ConcreteFortranSymmetricPackedArray2d.C

```
FortranSymmetricPackedSubscriptor::ProjectionT::
ProjectionT(const FortranSymmetricPackedSubscriptor& shape,
            Dimension projected_out,
            Subscript s
            ) :
    proj_sub(s), d(projected_out), n(shape.shape(0)) {
}
```

and it is used in the offset computation:

Array/ConcreteFortranSymmetricPackedArray2d.C

```
Subscript
FortranSymmetricPackedSubscriptor::ProjectionT::
offset(const SubscriptArray<1>& s) const {
    if (d == 1) {
        // Offset within column; columns stored contiguously
        Subscript i = s(0);
        return (i <= proj_sub) ? i : ( i*(i+1) - proj_sub*(proj_sub-1) ) / 2;
    }
    else {
        // Offset within row
        Subscript j = s(0);
        return (proj_sub <= j) ? ( j*(j+1) - proj_sub*(proj_sub-1) )/2 : j;
    }
}
```

This example illustrates the benefits of the work we did in Chapter 13 to build a system of concrete array classes parameterized by subscriptor: Writing a class template for symmetric packed arrays only required defining a few trivial shape functions and the offset computations for the array and its projections.

18.6 Matrices Implemented with BLAS

The widely used BLAS subroutine libraries form the basis of many modern numerical libraries. BLAS comes in three levels: BLAS-1 [69] for vector-vector operations, BLAS-2 [41] for matrix-vector operations, and BLAS-3 [42] for matrix-matrix operations. In this section, we use the BLAS and our concrete FORTRAN-compatible arrays to implement matrices.

18.6.1 Wrapping BLAS

We wrap BLAS-3 for use in C++ by essentially the same process we used for wrapping LAPACK, beginning with FORTRAN extern declarations (not shown) wrapped in overloaded static functions:

LapackWrap/BlasSubroutines.h

```
class Blas3Subroutines :
    public Blas2Subroutines {
public:
    static char trans_char[ ];
    static void xgemm(
        Trans t_a,
        Trans t_b,
        int m, int n, int k,
        double alpha, const double* a_p, int lda,
        const double* b_p, int ldb, double beta,
        double* c_p, int ldc
    );
    static void xgemm(
        Trans t_a,
        Trans t_b,
        int m, int n, int k,
        float alpha, const float* a_p, int lda,
        const float* b_p, int ldb, float beta,
        float* c_p, int ldc
    );
    // ...
};
```

The base class Blas2Subroutines contains static wrapper functions for BLAS-2; it derives from Blas1Subroutines, which contains static wrapper functions for BLAS-1. (Neither base class is shown.) Static member functions are inherited from base classes, so each successive level of BLAS wrapper adds additional functions: Blas3Subroutines provides the functions for BLAS-3, BLAS-2, and BLAS-1.

18.6.2 Matrices

The first step in writing our matrix class is to figure out what kind of algebraic structure to give it. Using Fig. 16.1, we can see that matrices with floating point elements form a division algebra under matrix addition, subtraction, multiplication, and inversion, with scalar multiplication by floating point numbers. However, even though (nonsingular) matrices are invertible, BLAS-3 does not provide matrix inversion. Therefore we will endow our BLAS-based matrix class, called ConcreteBlas2d<T>, with the structure of an algebra with unit instead of a division algebra.

We'll use the distributing arithmetic classes of Section 16.6 to implement its user-must-define functions for an Abelian group and BLAS to implement its other arithmetic:

LapackWrap/ConcreteBlas2d.h

```
template<class T>
class ConcreteBlas2d :
    public AlgebraWithUnitCategory<ConcreteBlas2d<T>, T>,
    public DistributingAbelianGroup<ConcreteBlas2d<T>, T>,
    public ConcreteFortranArray2d<T> {
public:
    ConcreteBlas2d(const ConcreteFortranArray2d<T>& a);
    ConcreteBlas2d(Subscript nrows, Subscript ncols);

    typedef ConstConcreteBlasProjection1d<T> ConstProjectionT;
    typedef      ConcreteBlasProjection1d<T>      ProjectionT;

    // AlgebraWithUnitCategory ops not implemented by DistributingAbelianGroup.
    ConcreteBlas2d<T>& operator*=(const T& rhs);
    ConcreteBlas2d<T>& operator*=(const ConcreteBlas2d<T>& rhs);
    ConcreteBlas2d<T>& operator/=(const T& rhs);
    ConcreteBlas2d<T>& setToOne();

    ConcreteBlas2d<T>& operator=(const ConcreteBlas2d<T>& rhs);
    ConcreteBlas2d<T>& operator=(const T& rhs);

    ConstProjectionT project(Subscript i, Dimension d = 0) const;
          ProjectionT project(Subscript i, Dimension d = 0);
```

```
ConstProjectionT operator[](Subscript i) const { return project(i, 0); }
      ProjectionT operator[](Subscript i)       { return project(i, 0); }

ConstProjectionT row     (Subscript i) const { return project(i, 0); }
      ProjectionT row     (Subscript i)       { return project(i, 0); }
ConstProjectionT column(Subscript i) const { return project(i, 1); }
      ProjectionT column(Subscript i)       { return project(i, 1); }

    // Matrix – Vector (Blas Level 2) Operation
friend ConcreteBlas1d<T>
                operator*(const ConcreteBlas2d<T>& m, const ConcreteBlas1d<T>& v);
};
```

The constructors and assignment operators call the corresponding functions in the ConcreteFortranArray2d<T> base; they are necessary because constructors and assignment operators are not inherited. The fundamental projection functions, project(), are discussed in Section 18.6.3; the other projection functions all call project() with specific parameters, as in all of our two-dimensional array classes.

The arithmetic computations are implemented by calling BLAS subroutines. Here are two examples:

LapackWrap/ConcreteBlas2d.c

```
template<class T>
ConcreteBlas2d<T>& ConcreteBlas2d<T>::operator*=(const T& rhs) {
    Blas3Subroutines::xscal(numElts(), rhs, firstDatum(), 1);
    return *this;
}

template<class T>
ConcreteBlas1d<T>
operator*(const ConcreteBlas2d<T>& m, const ConcreteBlas1d<T>& v) {
    ConcreteBlas1d<T> result( m.shape(0) );
    Blas3Subroutines::xgemv(
        Blas3Subroutines::no_trans,
        m.shape(0), m.shape(1),
        T(1), m.firstDatum(), m.shape(0),
        v.firstDatum(), 1,
        T(0), result.firstDatum(), 1
        );
    return result;
}
```

The ConcreteBlas1d<T> class template represents vectors in a linear space compatible with ConcreteBlas2d<T>; see Exercise 18.11.

The matrix-matrix operator*=() function can't be implemented as an in-place operation because the shape of the product matrix can be different from the shape

of the original left-hand matrix. Thus we have to copy the left-hand side and adjust the size of the matrix before computing the product with a call to BLAS:

LapackWrap/ConcreteBlas2d.c

```
template<class T>
ConcreteBlas2d<T>& ConcreteBlas2d<T>::operator*=(const ConcreteBlas2d<T>& rhs) {
    ConcreteBlas2d<T> lhs(*this);    // Copy out left operand
    reshape(SubscriptArray<2>(shape(0), rhs.shape(1)));    // Make room for product
    Blas3Subroutines::xgemm(
        Blas3Subroutines::no_trans, Blas3Subroutines::no_trans,
        lhs.shape(0), rhs.shape(1), rhs.shape(0),
        T(1), lhs.firstDatum(), lhs.shape(0),
        rhs.firstDatum(), rhs.shape(0),
        T(0), firstDatum(), shape(0)
    );
    return *this;
}
```

Although convenient from an implementation standpoint, this copying is unnecessary; see Exercise 18.15.

18.6.3 Matrix Projections

Projections of `ConcreteBlas2d<T>` objects yield `ConcreteBlasProjection1d<T>` objects (not shown) or their `Const` counterparts:

LapackWrap/ConcreteBlas2d.h

```
template<class T>
class ConstConcreteBlasProjection1d :
    public ConcreteFortranArray2d<T>::ConstProjectionT {
public:
    ConstConcreteBlasProjection1d(
            const ConcreteFortranArray2d<T>::ConstProjectionT& underlying_proj) :
        ConcreteFortranArray2d<T>::ConstProjectionT(underlying_proj) {}
    ConstConcreteBlasProjection1d(
            ConcreteFortranArray2d<T>::ConstProjectionT::SubscriptorT s, const T* p) :
        ConcreteFortranArray2d<T>::ConstProjectionT(s, p) {}

    friend ConcreteBlas1d<T>
        operator*(const ConstConcreteBlasProjection1d<T>& lhs, const T& rhs);
    friend ConcreteBlas1d<T>
        operator*(const T& lhs, const ConstConcreteBlasProjection1d<T>& rhs);
    friend ConcreteBlas1d<T>
        operator/(const ConstConcreteBlasProjection1d<T>& lhs, const T& rhs);

    double dot(const ConstConcreteBlasProjection1d<T>& rhs) const;
```

```
private:
    // Copying a projection would just copy the reference to the underlying array,
    // not the elements. To avoid confusion, we prohibit copying.
    ConstConcreteBlasProjection1d(const ConstConcreteBlasProjection1d<T>&);
};
```

These projections are the projections of ConcreteFortranArray2d<T>, augmented with appropriate vector-vector and vector-scalar operations. The constructors are used to initialize the base subobject, as in the ConcreteBlas2d<T> project() function:

LapackWrap/ConcreteBlas2d.c

```
template<class T>
ConstConcreteBlasProjection1d<T>
ConcreteBlas2d<T>::project(Subscript i, Dimension d) const {
    return ConcreteFortranArray2d<T>::project(i, d);
}
```

The vector-vector and vector-scalar operations are implemented using BLAS-1, either directly or through ConcreteBlas1d<T> (not shown). For example, the dot() function is implemented directly:

LapackWrap/ConcreteBlas2d.c

```
template<class T>
double
ConstConcreteBlasProjection1d<T>::
dot(const ConstConcreteBlasProjection1d<T>& rhs) const {
    if ( shape(0) != rhs.shape(0) ) throw ArrayErr::Shape();
    return
      Blas1Subroutines::xdot(shape(0), firstDatum(), offset(1), rhs.firstDatum(), rhs.offset(1));
}
```

The BLAS-1 xdot subroutine was written to work with vectors having strides other than 1, specifically to allow it to be used on matrix projections. We obtain the stride information by computing the offset of the second element and subtracting from it the offset of the first element; since the latter is always zero, the subtraction can be omitted. The vector-scalar operations are implemented indirectly like this:

LapackWrap/ConcreteBlas2d.c

```
template<class T>
ConcreteBlas1d<T>
operator*(const ConstConcreteBlasProjection1d<T>& lhs, const T& rhs) {
    ConcreteBlas1d<T> result(lhs);
    return result *= rhs;
}
```

18.6.4 Matrix Transposes

We conclude our discussion of matrices with a class to support the product with transposed matrices that occurs commonly in matrix expressions. We could offer a transpose() member function in ConcreteBlas2d<T>, but the data movement overhead could be excessive for large matrices. Instead TransposedConcreteBlas2d<T> refers to an existing matrix much as a projection does:

LapackWrap/TransposedConcreteBlas2d.

```
template<class T>
class TransposedConcreteBlas2d :
    public ConcreteArray2dRef<ConcreteRowMajorSubscriptor<2>, T> {
public:
    TransposedConcreteBlas2d(ConcreteBlas2d<T>& a);

    friend ConcreteBlas2d<T>
        operator*(const TransposedConcreteBlas2d<T>& t,
            const TransposedConcreteBlas2d<T>& u);
    friend ConcreteBlas2d<T>
        operator*(const ConcreteBlas2d<T>& u,
            const TransposedConcreteBlas2d<T>& t);
    friend ConcreteBlas2d<T>
        operator*(const TransposedConcreteBlas2d<T>& t, const ConcreteBlas2d<T>& u);
    friend ConcreteBlas1d<T>
        operator*(const TransposedConcreteBlas2d<T>& t, const ConcreteBlas1d<T>& u);

    operator ConcreteBlas2d<T>() const;
};
```

Objects in this class are like $\mathbf{U}^T$ for ConcreteBlas2d<T> representations of $\mathbf{U}$. The tranposition is obtained by changing the subscriptor: ConcreteBlas2d<T> is implemented with a column-major subscriptor (via ConcreteFortranArray2d<T>), whereas TransposedConcreteBlas2d<T> is implemented with a row-major subscriptor. The same data are accessed differently in the two cases.

The only operations applicable to TransposedConcreteBlas2d<T> objects are creation from ConcreteBlas2d<T> objects, the three possible matrix products, a matrix-vector product, and conversion to a ConcreteBlas2d<T>. The constructor initializes the array reference with a subscriptor and a pointer to the data:

LapackWrap/TransposedConcreteBlas2d.c

```
template<class T>
TransposedConcreteBlas2d<T>::TransposedConcreteBlas2d(ConcreteBlas2d<T>& a) :
    ConcreteArray2dRef<ConcreteRowMajorSubscriptor<2>, T>(
        SubscriptArray<2>(a.shape(1), a.shape(0)),    // Subscriptor initializer
        a.firstDatum()
    ) {
}
```

The subscriptor is created implicitly from the SubscriptArray<2>, which represents the shape of the transposed matrix.

The matrix products follow this pattern as well:

LapackWrap/TransposedConcreteBlas2d.c

```
template<class T>
ConcreteBlas2d<T> operator*(const TransposedConcreteBlas2d<T>& t,
                            const TransposedConcreteBlas2d<T>& u
                           ) {
    ConcreteBlas2d<T> result(t.shape(0), u.shape(1));
    Blas3Subroutines::xgemm(
        Blas3Subroutines::trans, Blas3Subroutines::trans,
        t.shape(0), u.shape(1), t.shape(1),
        T(1), t.firstDatum(), t.shape(1),
        u.firstDatum(), u.shape(1),
        T(0), result.firstDatum(), result.shape(0)
    );
    return result;
}
```

and the conversion copies data:

LapackWrap/TransposedConcreteBlas2d.c

```
template<class T>
TransposedConcreteBlas2d<T>::operator ConcreteBlas2d<T>() const {
    ConcreteBlas2d<T> result(shape(1), shape(0));
    concreteCopy(result, ConcreteArray2dConstRef<SubscriptorT, T>(*this));
    return result;
}
```

We'll see an example of how this transpose capability can be used in the next section.

18.7 Singular Value Decomposition with LAPACK

In Chapter 15, we illustrated the conceptual relation between class and subroutine libraries with LAPACK's subroutines for the familiar LU decomposition of matrices. Then, in Section 18.4, we saw how to make the connection to LAPACK, calling LAPACK FORTRAN subroutines to implement the design from Chapter 15. The resulting class is suitable for solving small, well-conditioned systems of linear equations. However, for larger or ill-conditioned systems, the more modern *singular value decomposition* (*SVD*) algorithm is better. We employ SVD in Chapter 19. Having tackled the simpler LU decomposition, we now extend the system of classes developed in Chapter 15 to incorporate LAPACK's SVD subroutines. These classes build on the matrix classes developed in the preceding section.

18.7.1 Singular Value Decomposition

For square, nonsingular $\mathbf{A}$, the solution for $\mathbf{Ax} = \mathbf{b}$ is $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. To obtain reasonable solutions for $\mathbf{Ax} = \mathbf{b}$ for more general $\mathbf{A}$, singular value decomposition extends the idea of matrix inverse to singular and nonsquare matrices. Given an $m \times n$ matrix $\mathbf{A}$, there exists an $m \times m$ orthogonal matrix $\mathbf{U}$, an $m \times n$ matrix $\mathbf{\Sigma}$, and an $n \times n$ orthogonal matrix $\mathbf{V}$ such that

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{18.1}$$

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_r & & \\ & & & 0 & \\ & & & & \ddots \end{pmatrix}, \tag{18.2}$$

where $r = \min(m, n)$. The first r columns of $\mathbf{U}$ and $\mathbf{V}$ are called, respectively, the *left singular vectors* and *right singular vectors*, and the σ_i, ordered such that $\sigma_1 \geq \cdots \geq \sigma_r \geq 0$, are called the *singular values*.

When $\mathbf{A}$ is square and nonsingular, its factored form $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$ can be inverted easily: $\mathbf{U}$ and $\mathbf{V}$ are orthogonal, so $\mathbf{U}^T\mathbf{U} = \mathbf{I}$ and $\mathbf{V}^T\mathbf{V} = \mathbf{I}$, and $\mathbf{\Sigma}$ is $m \times m$ with $r = m$. Therefore $\mathbf{A}^{-1} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^T$.

When $\mathbf{A}$ is singular or nonsquare, $\mathbf{\Sigma}$ cannot be inverted because it has some zero values on the diagonal. However, an $n \times m$ *pseudoinverse* matrix [70, Chapter 7], denoted $\mathbf{A}^+$, can be defined such that $\mathbf{A}\mathbf{A}^+ = \mathbf{A}^+\mathbf{A} = \mathbf{I}$. Letting

$$\mathbf{\Sigma}^+ = \begin{pmatrix} \sigma_1^{-1} & & & & \\ & \ddots & & & \\ & & \sigma_r^{-1} & & \\ & & & 0 & \\ & & & & \ddots \end{pmatrix},$$

it is easy to verify that

$$\mathbf{A}^+ = \mathbf{V}\mathbf{\Sigma}^+\mathbf{U}^T.$$

Each nonzero element of $\mathbf{\Sigma}$ is inverted in the corresponding element of $\mathbf{\Sigma}^+$; zero elements of $\mathbf{\Sigma}$ are set to zero in $\mathbf{\Sigma}^+$. When $\mathbf{A}$ is square and nonsingular, $\mathbf{A}^{-1} = \mathbf{A}^+$.

The pseudoinverse also solves the linear least squares problem: Given $\mathbf{A}$ and an m-vector $\mathbf{b}$, find the n-vector $\mathbf{x}$ that minimizes $\|\mathbf{Ax} - \mathbf{b}\|$, the Euclidean length of the residual vector $\mathbf{Ax} - \mathbf{b}$. One solution to this least squares problem is $\mathbf{x} = \mathbf{A}^+\mathbf{b}$.

In general, the least squares problem has many solutions, one for each way of assigning values to the elements of $\mathbf{\Sigma}^{-1}$ that correspond to zero elements in $\mathbf{\Sigma}$. Using the zero assignment adopted in $\mathbf{\Sigma}^{+}$ gives the unique solution with minimum norm, $||x||_{min}$.

The SVD is also useful when solving a system of equations with an almost singular matrix. In such cases, LU decomposition in floating point arithmetic yields an inaccurate solution. How close a matrix is to being singular is measured by its *condition number*, the ratio of its largest and smallest singular values; an ill-conditioned matrix has a large condition number. With a large condition number, least squares solutions to $\mathbf{Ax} = \mathbf{B}$ become dominated by roundoff error or noise. The solution can be improved by approximating $\mathbf{A}$ with a matrix with lower condition number. From Eq. (18.1),

$$A_{ij} = \sum_{k=1}^{n} \sigma_k U_{ik} V_{jk}. \tag{18.3}$$

Zeroing some small σ_i's yields a matrix $\tilde{\mathbf{A}}$ that is close to $\mathbf{A}$ but with a smaller condition number:

$$\tilde{\mathbf{\Sigma}} = \begin{pmatrix} \sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_p & & \\ & & & 0 & \\ & & & & \ddots \end{pmatrix}, p \leq r \tag{18.4}$$

$$\tilde{\mathbf{A}} = \mathbf{U}\tilde{\mathbf{\Sigma}}\mathbf{V}^T. \tag{18.5}$$

Now solve $\tilde{\mathbf{A}}\mathbf{x} = \mathbf{b}$ for x. The cutoff index p compromises between improving the condition of the matrix and increasing the residual $\|\mathbf{Ax} - \mathbf{b}\|$. See [70, Section 25.6] or [93, Section 2.6]. The choice of p is somewhat problem- and machine dependent. One rule of thumb is to zero all singular values less than $n\epsilon\sigma_1$, where ϵ is the machine precision [93, Section 15.4]. See also [50, Section 5.5.8].

18.7.2 Classes for Singular Value Decomposition

We now extend the classes from Chapter 15 to provide a representation that supports SVD factorization and its use for solving the linear least squares problem. It is rarely necessary to compute the pseudoinverse explicitly, so the factor-solve approach we used with LU decomposition is appropriate: Given the system $\mathbf{Ax} = \mathbf{b}$, the factor step computes $\mathbf{U}$, $\mathbf{\Sigma}$, and $\mathbf{V}^T$ and the solve step computes x as $\mathbf{V}\tilde{\mathbf{\Sigma}}^{+}\mathbf{U}^T\mathbf{b}$.

We choose a FORTRAN representation and use LAPACK for the SVD computation; the class parallels RectLURep<T> from page 466:

LapackWrap/RectSVDRep.h

```
template<class T>
class RectSVDRep {
public:
    typedef ConcreteBlas2d<T> Unfactored;      // The type of A in Ax = b;
    typedef ConcreteBlas2d<T> Unknowns2d;      // The type of x in Ax = b;
    typedef ConcreteBlas2d<T> Knowns2d;        // The type of b in Ax = b;
    typedef ConcreteBlas1d<T> Unknowns1d;      // The type of x in Ax = b;
    typedef ConcreteBlas1d<T> Knowns1d;        // The type of b in Ax = b;

    class Factored {
    public:
        Factored(RectSVDRep<T>::Unfactored* mp);
        void solve(Knowns1d& b);  // Overwrites b with x
        void solve(Unknowns1d& x, const Knowns1d& b);
        void solve(Knowns2d& b);  // Overwrites b with x
        void solve(Unknowns2d& x, const Knowns2d& b);

        Knowns1d&    singularValues()        { return sigma; }
        Unknowns2d& rightSingularVectorsT() { return v_T;   }
        Knowns2d&    leftSingularVectors()   { return u;     }
    protected:
        virtual void pseudoInvert(Knowns2d& uTb);    // U^T * b * sigma^+
        virtual void pseudoInvert(Knowns1d& uTb);

        virtual T zeroingTol() const;                // Tolerance for zeroing sigma_i

                                  // Factored matrix pointer
        CopiedObjPtr<RectSVDRep<T>::Unfactored> facmat_p;
        Subscript     m;          // Number of rows in A
        Subscript     n;          // Number of columns in A
        Subscript     k;          // min(m,n), size of matrices.
        Knowns1d      sigma;      // Singular values
        Unknowns2d v_T;           // Right singular vectors, transposed
        Knowns2d      u;          // Left singular vectors
    };
};
```

New public functions added to the Factored class allow manipulation of the factors. New protected virtual functions control the formation of $\mathbf{\Sigma}^+$ during the solve()

step; derived classes can alter the effective Σ^+ used in solve() by overriding these functions (see page 604). Note that Σ is stored in a one-dimensional array holding just the diagonal elements.

Implementing RectSVDRep<T> factoring amounts to supplying the appropriate arguments and work space for LAPACK's xgesvd() function:

LapackWrap/RectSVDRep.c

```
template<class T>
RectSVDRep<T>::Factored::Factored(RectSVDRep<T>::Unfactored* mp) :
    facmat_p(mp),
    m( mp->shape(0) ),
    n( mp->shape(1) ),
    k( min(m, n) ),
    sigma(k),                   // k X k diagonal matrix, stored as vector
    u(m, k),                    // m X k
    v_T(k, n) {                 // k X n

    // Minimum size work area required by LAPACK
    ConcreteFortranArray1d<T> work( max(3 * k + max(m,n), 5 * k - 4) );

    int info;    // info return from LAPACK factor routine.
    LapackSubroutines::xgesvd(
        LapackSubroutines::separate,    // Give just left singular vectors, in u.
        LapackSubroutines::separate,    // Give just right singular vectors, in v_t.
        m, n, mp->firstDatum(), mp->shape(0),       // A
        sigma.firstDatum(),                         // sigma's
        u.firstDatum(), u.shape(0),                 // U
        v_T.firstDatum(), v_T.shape(0),             // V^T
        work.firstDatum(), work.numElts(),          // Work area
        info                                        // Return code
    );
    if (info != 0) throw LapackErr::UnableToFactorSVD(info);
}
```

The solve step is implemented straightforwardly using the TransposedConcreteBlas2d<T> class (page 568) to multiply by the transposed orthogonal matrices:

LapackWrap/RectSVDRep.c

```
template<class T>
void RectSVDRep<T>::Factored::solve(RectSVDRep<T>::Knowns2d& b) {
    b = TransposedConcreteBlas2d<T>(u) * b;         // U^T * b
    pseudoInvert(b);                                // sigma~^ + * b
    b = TransposedConcreteBlas2d<T>(v_T) * b;       // V * b
}
```

Recall that the transpose is only a different access to the matrix. The Σ^+ is also not formed explicitly; it only enters via the pseudoInvert() algorithm:

LapackWrap/RectSVDRep.c

```
template<class T>
void RectSVDRep<T>::Factored::pseudoInvert(RectSVDRep<T>::Knowns2d& b) {
    T tol = zeroingTol();
    for (Subscript i = 0; i < k; i++) {
        if ( sigma(i) < tol ) b.row(i) = T(0);
        else b.row(i) *= T(1) / sigma(i);
    }
}
```

The threshold for zeroing reciprocals of singular values is expressed in terms of the precision of the floating point arithmetic used:

LapackWrap/RectSVDRep.c

```
template<class T>
T RectSVDRep<T>::Factored::zeroingTol() const {
    return NumericalLimits<T>::epsilon * n * sigma(0);
}
```

Since this, in turn, depends on the template parameter T, we express type-dependent floating point constants with a class template:

SciEng/NumericalLimits.h

```
template<class T>
class NumericalLimits {
public:
    static const T minimum;              // Minimum normalized positive T number.
    static const T maximum;              // Maximum normalized positive T number.
    static const T epsilon;              // Minimum x such that 1.0 + x != x.

    static const int digits;             // decimal digits of precision.
    static const int radix;              // exponent representation radix.
    static const int mantissa_digits;    // number of base radix digits in mantissa;

    static const int min_exponent;       // pow(radix,min_exponent - 1) is ok.
    static const int min_10_exponent;    // pow(10,min_10_exponent) is ok.
    static const int max_exponent;       // pow(radix,max_exponent - 1) is ok.
    static const int max_10_exponent;    // pow(10,max_10_exponent) is ok.
};
```

These constants are also machine dependent, so we use template specialization to set their values from constants defined in the standard header float.h:

SciEng/NumericalLimits.l

```
#include <float.h>

const float    NumericalLimits<float>::minimum          = FLT_MIN;
const float    NumericalLimits<float>::maximum          = FLT_MAX;
const float    NumericalLimits<float>::epsilon          = FLT_EPSILON;
// ...

const double NumericalLimits<double>::minimum           = DBL_MIN;
const double NumericalLimits<double>::maximum           = DBL_MAX;
const double NumericalLimits<double>::epsilon           = DBL_EPSILON;
// ...
```

This completes our work to provide singular value decomposition within the class framework we designed in Chapter 15. We now look at two simple example uses of the SVD before moving on to a more sophisticated example in Chapter 19.

18.7.3 Example: Least Squares Solution of a Linear System

To illustrate and test the SVD, we apply it to a test case described by Aki and Richards [3, Section 12.3.1]:

$$x_1 + x_2 = 1$$
$$x_3 = 2$$
$$-x_3 = 1.$$

The variables x_1 and x_2 are underdetermined and x_3 is overdetermined. Even though we have three equations and three unknowns, no solution is possible with conventional matrix inversion. The pseudoinverse solution is computed by the following code:

ch18/tRectSVDRep.C

```
// Example from Aki and Richards, Quantitative Seismology,
// Volume 2, Sections 12.3.1 and 12.3.2
LapackUnfactored< RectSVDRep<float> > a(3, 3);
a(0,0) = 1;      a(0,1) = 1;      a(0,2) =  0;
a(1,0) = 0;      a(1,1) = 0;      a(1,2) =  1;
a(2,0) = 0;      a(2,1) = 0;      a(2,2) = -1;

LapackFactored< RectSVDRep<float> > fa = a.factor();

ConcreteBlas1d<float> b(3);
b(0) = 1; b(1) = 2; b(2) = 1;

cout << fa.solve(b) << endl;
```

The output is

```
[0.5, 0.5, 0.5]
```

18.7.4 Example: Fitting to Legendre Polynomials

We now illustrate the use of the SVD classes for solving a simple linear least squares fitting problem: Fit a set of n data points (x_i, y_i) to a model that is a linear combination of the first five Legendre polynomials $P_0(x), \ldots, P_4(x)$ (cf. Exercise 17.7). In other words, we want to find the coefficients $c_0, \ldots, c_4$ of

$$y(x) = \sum_{l=0}^{4} c_l P_l(x)$$

that minimize

$$\sum_{i=0}^{n-1} (y_i - y(x_i))^2.$$

The equivalent matrix formulation is

$$\begin{pmatrix} P_0(x_0) & \cdots & P_4(x_0) \\ \vdots & \vdots & \vdots \\ P_0(x_{n-1}) & \cdots & P_4(x_{n-1}) \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_{n-1} \end{pmatrix}. \tag{18.6}$$

The following test code generates data points by evaluating the polynomial $x^3 + x^2 + x + 1$ at the integers from 0 to 10, setting up Eq. (18.6):

ch18/tLegendreFit

```
LapackUnfactored< RectSVDRep<double> > P(11, 5);
RectSVDRep<double>::KnownsId y(11);

for (int i = 0; i <= 10; i++) {
    // Set up row i of matrix P_{il} = P_l(x_i)
    int isq = i * i;
    P(i, 0) = 1.0;
    P(i, 1) = i;
    P(i, 2) = 1.5 * isq - .5;
    P(i, 3) = i * (2.5 * isq - 1.5);
    P(i, 4) = .125 * (isq * (35 * isq - 30) + 3);

        // Compute the right-hand side y_i = i^3 + i^2 + i + 1
    y(i) = ((i+1) * i + 1) * i + 1;
}
```

```
// Solve
RectSVDRep<double>::Unknowns1d result(5);
cout << P.factor().solve(result, y) << endl;
```

The values of the Legendre polynomials are computed by evaluating the polynomials directly; see Exercise 18.18 for a more flexible approach. This test code prints

```
[1.33333, 1.6, 0.666667, 0.4, -2.76198e-17]
```

The last value is small, as we would expect when fitting a fourth-degree polynomial to data generated from a third-degree polynomial.

18.8 Summary

In this chapter, we explored ways of using legacy code, either directly through function calls or indirectly through classes that wrap function calls and data in objects. Wrapping existing libraries illustrates many of the goals of object-oriented programming. Our wrappers don't do more computations than the original code. They reorganize the computation and repackage it. Examined closely and in detail as we have done here, the work of designing classes and coding them seems large. But the point of wrapping classes around existing libraries is to make the future use of the libraries easier and more robust.

This general theme—investing design work up front for improvements in later use—echoes the original aim of these subroutine libraries. Subroutine libraries package algorithms but work on built-in types; class libraries create new types that package various built-in types with algorithms. Moreover, class libraries can express more complex couplings inside of simpler boxes than can subroutine libraries. By working hard to hide the details, we can turn our attention to more complex problems.

18.9 Notes and Comments

18.1 Teale [111, Chapter 3] discusses the advantages of using the iostream classes for input/output instead of C's stdio functions.

18.2 Experience in wrapping a large, interactive, computer-aided design system written in PL/1 with objects is reported in [35, 36]. Unlike the subroutines we wrapped in this chapter, the computer-aided design system maintains state—in a sense it has its own objects, even though they don't appear in its code. State complicates the wrapper considerably.

18.3 Calling a function (subroutine) written in one language from another language requires, in general, addressing many details. These include argument passing conventions, function return values, memory management, exception handling, compatibility of input/output, and data representation. C++ linkage specifications do not address

any of these issues. In fact, linkage between two language implementations is only possible if their implementors made similar choices, as is often the case among compilers written by one vendor.

C++ linkage specifications primarily address naming issues that arise in most implementations. Most operating systems provide a linker that combines separately compiled object code into one executable module. A typical linker uses names to connect a function call in one part of the program to a function definition in another part. Since C++ functions are overloaded based on argument types, several functions can have the same name, a problem for the linker. Therefore many C++ implementations encode

M §7.2.1c argument type information in the function names generated in the object code, sometimes called *name mangling*. For example, a function f() with no arguments will have a different name than a function f() that takes a single int argument.

When the compiler generates a call to a function written in another language, the name can't be mangled as usual, or else the generated call won't match the corresponding function definition generated by the compiler for the other language. The principal effect of specifying a non-C++ linkage for a function is to suppress name mangling for that function.

18.4 The documentation that comes with most ANSI C compilers provides the calling sequences for the ANSI C string library. An excellent summary of the functions and examples of how they are used appear in [58, Chapter 13]. A detailed discussion of the library's design and implementation can be found in [91, Chapter 14].

18.5 An ANSI C++ standardization committee is working toward a standard C++ character string class. Our String class is unrelated to any of the several proposals and is intended primarily to illustrate wrapping of a C library. We have found it useful in the absence of the standard class.

18.6 The ANSI C++ standardization committee is considering introducing the notion of a *sequence point* into the language, modeled after sequence points in ANSI C. In ANSI C, a sequence point exists at the end of every expression that is not part of a larger expression and at a few other places. See [58, Section 4.4.5] for a description of C sequence points. If the notion is adopted for C++, temporary objects could not be destroyed except at sequence points, eliminating the subtle dangling reference problem associated with the c_str() members of String. See also [44, Section 12.2] for a discussion of the issues involved in choosing a time to destroy temporary objects.

18.7 The idea of using an operator constructor to eliminate unnecessary copying is discussed in [22, Chapter 7]. Using a constructor call as the return expression for a function is also discussed in [44, Section 12.1.1c].

18.10 Exercises

18.1 Implement the String::subString() member functions from Section 18.2.3.

18.2 Implement the member functions for the String browser and iterator classes declared on page 547.

18.3 Alter String to provide substrings that refer to the original string, much like array projections refer to the original array elements. Be sure that your solution handles const String objects correctly.

18.4 The statement cs += buf in the definition of operator>>(istream& , String&) on page 552 creates a temporary String initialized from buf, concatenates it with cs, and then destroys it. This results in unnecessary copying of the contents of buf. Modify the String class to avoid this copying; do not modify operator>>(istream& , String&).

18.5 The strcat() function in the ANSI C library concatenates the string pointed to by its second argument to the end of the string pointed to by its first argument. To do this, it must find the null terminator at the end of the first argument. Modify the String function definitions to avoid this search.

18.6 String does not provide all of the facilities provided by the ANSI C string library. One example is the strtok() function, which is useful for parsing character strings into tokens or words. You specify what characters are considered to separate tokens in the string and then call the function repeatedly to extract the tokens. Study the description of strtok() in your C compiler's manual (or other source, e.g., [91] or [58]), and then provide the equivalent capability for use with String. Pay careful attention to the effect of the state stored across multiple calls to strtok().

18.7 Since each (natural) language has its own character set and ordering among characters, ANSI C has the notion of a *locale*, a description of various characteristics that are affected by choice of language. The ANSI C string library has several functions whose results depend on the choice of locale. Study the description of the string library in your C compiler's manual (or other source, e.g., [91] or [58]), and then provide the equivalent function in String or a class associated with it.

18.8 It is often convenient to be able to do output into a String. For example, the BlasErr message() function on page 556 writes the number the_arg_num into the String s. Use the ostrstream class that is part of the iostream class library (see, e.g., [111] or your compiler's library manual) to provide a definition for the following function template:

SciEng/String.h

```
template<class T>
String& operator<<(String& s, const T& obj);    // Format into string object
```

18.9 Implement the constructor and solve() members of SymPosDefPackedLURep<T>::Factored (defined on page 467).

18.10 LAPACK stores an $n \times n$ banded matrix (see Figure 15.1 on page 454) with k_l subdiagonals and k_u superdiagonals in a two-dimensional array with $k_l + k_u + 1$ rows and n columns [5, Section 5.3.3]. For example, the matrix

$$\begin{matrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} \\ 0 & 0 & a_{53} & a_{54} & a_{55} \end{matrix}$$

would be stored like this:

$$\begin{array}{ccccc} & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & \\ a_{31} & a_{42} & a_{53} & & \end{array}$$

where blank entries need not be stored. (The storage savings are large if $k_l \ll n$ and $k_u \ll n$.) Design and implement a class to represent this storage layout.

18.11 Implement ConcreteBlas1d<T>, a class that forms a linear space with scalars of type T. Use derivation from ConcreteFortranArray1d<T> and DistributingLinearSpace<V,S>.

18.12 Extend ConcreteBlas2d<T> to include suitable vector-matrix arithmetic via calls to BLAS-2.

18.13 The definitions of the operator[ ](), row(), and column() member functions of ConcreteBlas2d<T> (page 564) are identical to the definitions of the corresponding functions of ConcreteArray2d<Subscriptor, T> (page 375). ConcreteArray2d<Subscriptor, T> is a base class of ConcreteBlas2d<T>. Why couldn't we just inherit these member functions?

18.14 ConcreteBlas2d<T> is derived publicly from ConcreteFortranArray2d<T> and therefore exhibits the schizophrenia described in Section 10.3. Discuss the advantages and disadvantages of this design.

18.15 ConcreteBlas2d<T> uses the algebraic structure categories to implement the * operator via its operator*=() member function. The operator*=() definition on page 566 copies and then reshapes the left-hand matrix before calling a BLAS subroutine to do the multiplication. Modify ConcreteBlas2d<T> or one of its bases to make it possible to eliminate this copying. Identify additional unnecessary copies in the * operator provided by the algebraic structure categories. Attempt to use template specialization to eliminate these copies and discuss the problems you encounter.

18.16 In Exercise 18.15, you worked to eliminate unnecessary copies. Is the cost of these copies significant? Discuss.

18.17 A linear system $A\mathbf{x} = \mathbf{b}$ can be transformed by change of variable into the problem $\tilde{A}\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$, where $\tilde{A} = AH$, $\tilde{\mathbf{b}} = \mathbf{b}$, and H is nonsingular. Given the solution $\tilde{\mathbf{x}}$ of the transformed problem, the solution of the original problem is $\mathbf{x} = H\tilde{\mathbf{x}}$. When H is diagonal, the transformation consists of scaling the columns of A. Lawson and Hanson [70, Section 25.3] suggest using the column scaling transformation to improve the conditioning of the problem by setting

$$H_{jj} = \begin{cases} \|A_j\|^{-1} & \text{if } \|A_j\| \neq 0 \\ 1 & \text{if } \|A_j\| = 0 \end{cases},$$

where $\|A_j\|$ denotes the Euclidean norm of the jth column vector of A.

Implement two class templates, NormalizedLapackUnfactored<Rep> and NormalizedLapackFactored<Rep>, that apply this transformation before factoring and again after solving the transformed system to recover the solution to the original problem. Test your code with both LU decomposition and SVD, using both the RectLURep<T> and RectSVDRep<T> representations.

18.18 The code on page 576 fits a set of artificially generated data points to a model that is a linear combination of the first five Legendre polynomials. Rewrite this code to read from the input file the number of Legendre polynomials to use and then an arbitrary number of data points. Use your solution to Exercise 17.7 to compute the values of the Legendre polynomials.

CHAPTER 19

Data Modeling in C++

19.1 Introduction

In this concluding chapter, we develop software that solves a practical problem in scientific computing: nonlinear parameter estimation for data modeling, sometimes called peak fitting. In the process we illustrate how a system of classes grows to match and solve a problem.

Peak fitting and similar parameter estimation problems occur frequently in science and engineering. Such problems begin with a model of the physical process that we expect can reproduce experimental data. For example, geophysical models attempt to reproduce seismic readings on the earth's surface. Or physical models of photoelectron excitation attempt to reproduce measurements of electron intensity versus electron energy.

Once a model has been tested and shown to reproduce a wide variety of data, we can abstract the general parts of the model from its particular parts. For example, different rock-density profiles give rise to different seismic readings with a given model for the relation of rock density and excitation to seismic activity. Or different chemical states of atoms in a sample give different photoelectron spectra using the same model for mapping chemical state to photoelectron intensity. Parameter estimation is an attempt to infer particular information from particular measurements combined with a model of the relevant physical processes. Data modeling combines measured information—the data—with assumptions or *a priori* information—the model—producing interpreted or reduced data.

The code developed in this chapter uses many of our previous examples. We will work with experimental data, numerical models, and iterative solutions of linear equations. For the experimental data, we use a simple interface adaptable to various data formats. We require partial derivatives of the numerical model; we obtain these by *automatic differentiation,* an algebraic technique that builds on the work in Chapter 16. For the iterative solutions of linear equations, we extend the iterators of Chapter 13 and the LAPACK examples of Chapter 15. Along the way

we exploit the array classes from Chapter 13 and the matrix arithmetic from Section 18.6. Much of the computation is ultimately done by calls to the existing BLAS and LAPACK subroutine libraries via the wrappers described in Chapter 18.

19.2 Classes for Experimental Data

As a computing problem, data modeling works with measured data, a computational model, and an algorithm to adjust parameters in the model to fit the data. We start with the data.

Physical data represent measurements of some quantity in a physical system as another quantity is varied. For example, seismic-activity monitoring records earth-position changes versus time; photoelectron spectra record electron-emission intensity versus electron energy. The data are expected to have measurement errors. The combination of the measured quantity y, the varied quantity x, and the error σ is commonly called a *data point*.

The use and the representation of the data can be separated. To use a set of data points, we need to access the triples $\{y_i, x_i, \sigma_i\}$, $i = 1, n$, where n is the number of data points. We need not access all three components as an object nor all n data points at once.

The data could be represented using three arrays, or one array of three-component elements, or, if the varied quantity has regular steps and the errors are estimated from the measurements, one array of y_i's and two functions. Each of these representations could support all of the uses we need. Object-oriented programming encourages us to separate the use of the data points from their representation: Our estimation program should be written to use data from an interface base class without requiring a particular data representation.

We chose to provide two interfaces. One, called PhysicalDataBrowser<T>, gives read-only access to data point components, with functions value() for y_i, coordinate() for x_i, and sigma() for σ_i:

DataModeling/PhysicalData.h

```
template<class T>
class PhysicalDataBrowser {
public:
    virtual T coordinate() const = 0;
    virtual T value()      const = 0;
    virtual T sigma()      const = 0;

    virtual Boolean more()    const = 0;
    virtual void    advance() = 0;
    virtual ~PhysicalDataBrowser() {}
};
```

The interface is parameterized by the type T of the values, and the coordinates and sigmas are assumed to be of the same type.

The second interface provides a function returning an accessor to objects in the PhysicalDataBrowser interface category and the function numElts(), providing the number of data points:

DataModeling/PhysicalData.h

```
template<class T>
class PhysicalData {
public:
    virtual AccessedPhysicalDataBrowser<T> browser() const = 0;
    virtual Subscript                      numElts() const = 0;
    virtual ~PhysicalData() {}
};
```

This is the interface we pass to clients of our data readers; it allows us to get a new iterator over our data points whenever that is required.

A simple peak-fitting program might implement PhysicalData<T> with three arrays, with the arrays read from a file:

DataModeling/FormedPhysicalData.h

```
class FormedPhysicalData :
    public virtual PhysicalData<double> {
public:
    virtual AccessedPhysicalDataBrowser<double> browser() const {
        return new FormedPhysicalDataBrowser(x, y, sigma);
    }
    virtual Subscript numElts() const { return x.numElts(); }

    FormedPhysicalData(istream& stream_in, unsigned int n_data_pts);
    FormedPhysicalData(
        const ConstArray1d<double>& x_in,
        const ConstArray1d<double>& y_in,
        const ConstArray1d<double>& sigma_in
    );
    // ...
private:
    typedef ConcreteFormedArray1d<double> ArrayImpl;
    ArrayImpl x;
    ArrayImpl y;
    ArrayImpl sigma;
};
```

The constructor from a file throws an exception (class not shown) if too few data points can be read:

DataModeling/FormedPhysicalData.C

```
FormedPhysicalData::
FormedPhysicalData(istream& ns, unsigned int num_data_pts) :
    x(num_data_pts),
    y(num_data_pts),
    sigma(num_data_pts) {
    for (int n = 0; ns && n < num_data_pts; n++) {
        ns >> x(n) >> y(n) >> sigma(n);
    }
    if (n < num_data_pts) throw TooLittleDataInput();
}
```

The test of the stream object ns succeeds if the stream is still capable of providing more input.

The browser for FormedPhysicalData combines iterators over each array:

DataModeling/FormedPhysicalData.h

```
class FormedPhysicalDataBrowser :
    public virtual PhysicalDataBrowser<double> {
public:
    virtual double  coordinate() const { return x_i.current();     }
    virtual double  value()      const { return y_i.current();     }
    virtual double  sigma()      const { return sigma_i.current(); }

    virtual Boolean more()       const { return x_i.more();        }
    virtual void    advance()    { x_i.advance(); y_i.advance(); sigma_i.advance(); }

    FormedPhysicalDataBrowser(
            const ConcreteFormedArray1d<double>& x,
            const ConcreteFormedArray1d<double>& y,
            const ConcreteFormedArray1d<double>& sigma
        ) :
        x_i(x), y_i(y), sigma_i(sigma) {
    }
private:
    typedef ConcreteFormedArray1d<double> ArrayImpl;
    ArrayImpl::BrowserType x_i;
    ArrayImpl::BrowserType y_i;
    ArrayImpl::BrowserType sigma_i;
};
```

The accessor for FormedPhysicalDataBrowser, AccessedPhysicalDataBrowser<double>, is similar to other accessors we have shown (page 404) and is left as an exercise (Exercise 19.1).

19.3 Classes for Linearized Nonlinear Equations

In addition to the data points, $\{y_i, x_i, \sigma_i\}$, $i = 1, n$, we will also be given a *model*, a function of x_i that also depends on m adjustable parameters $\mathbf{a} = (a_0, \ldots, a_{m-1})$. We write the model as the function $y(x_i; \mathbf{a})$. For example, in photoelectron spectroscopy, with electron energies x_i and emission intensity $y_i \pm \sigma_i$, a model might be a sum of Gaussian functions:

$$y(x_i; \mathbf{a}) = a_0 e^{-(x_i - a_1)^2/a_2^2} + a_3 e^{-(x_i - a_4)^2/a_5^2}, \tag{19.1}$$

where $m = 6$: a_0 and a_3 are peak heights, a_1 and a_4 are peak positions, and a_2 and a_5 are peak widths.

Again, to separate use and definition, we write an interface base, `DataModel<T>`, for data models. Some characteristics of `DataModel<T>` are already evident: All data models act like a function taking coordinate values x_i and returning data-like values y, with some way to hold and set the parameters $\mathbf{a}$. This alone is not enough, however. To see why, we return to the mathematical development of our problem.

In principle, we should fit the model to the data by seeking the set of parameters $\mathbf{a}$ that has the *maximum likelihood* [93, Section 15.1]; in practice, we seek the set of parameters that minimizes the squared errors:

$$\chi^2(\mathbf{a}) = \sum_{i=1}^{n} \left(\frac{y_i - y(x_i; \mathbf{a})}{\sigma_i} \right)^2 . \tag{19.2}$$

This *chi-squared* or *least-squared error* criterion derives its popularity from its solvability. Moreover, excepting glitches, many distributions of experimental errors are close to normal or Gaussian distributions (see Exercise 19.7), in which case least squares fitting is a maximum likelihood fitting. Specifically, the value of many observations of a not-too-rare quantity will be normally distributed. Together these advantages are so great that nearly all maximum likelihood estimations use chi-squared fitting.

The least-squared error criterion makes rapid fitting of certain nonlinear models tractable. Those models that are suitable can be reasonably approximated by Taylor series expansions. The leading terms in the series will be linear in the model parameters. An approximate $\chi^2(\mathbf{a})$ formed from the linear terms will be a linear least squares problem. We saw how to solve linear least squares problems in Section 18.7. Minimizing $\chi^2(\mathbf{a})$ for nonlinear models proceeds by solving a linearized problem, using the result as a new point for Taylor series expansion, and iterating to convergence.

Expanding Eq. (19.2) in a Taylor series about an initial trial parameter set **z** gives

$$y(x_i; \mathbf{a}) \approx y(x_i; \mathbf{z}) + \sum_{j=1}^{m} \left(\frac{\partial y(x_i; \mathbf{a})}{\partial a_j} \right)_{\mathbf{a}=\mathbf{z}} (a_j - z_j). \tag{19.3}$$

Since the partial derivatives of the model with respect to **a** are evaluated at the initial parameter setting **z** (indicated by the subscript $\mathbf{a} = \mathbf{z}$), they are constants and the approximate model is linear. Using this in χ^2 gives

$$\chi^2(\mathbf{a}) \approx \sum_{i=1}^{n} \left(\frac{y_i - y(x_i; \mathbf{z}) - \sum_{j=1}^{m} \left(\frac{\partial y(x_i; \mathbf{a})}{\partial a_j} \right)_{\mathbf{a}=\mathbf{z}} (a_j - z_j)}{\sigma_i} \right)^2. \tag{19.4}$$

The Taylor expansion linearizes our problem, and we can write it as a linear least squares problem, minimizing $\chi^2(\mathbf{a}) = ||\mathbf{Ax} - \mathbf{b}||$ where

$$\mathbf{A}_{ij} = \frac{1}{\sigma_i} \left(\frac{\partial y(x_i; \mathbf{a})}{\partial a_j} \right)_{\mathbf{a}=\mathbf{z}}$$

$$\mathbf{x}_j = (a_j - z_j) \tag{19.5}$$

$$\mathbf{b}_i = \frac{y_i - y(x_i; \mathbf{z})}{\sigma_i}.$$

The model, $y(x_i; \mathbf{a})$, appears in two places, once evaluated at $\mathbf{a} = \mathbf{z}$ and once with its first partial derivatives evaluated at $\mathbf{a} = \mathbf{z}$.

Returning to the design of DataModel<T>, we represent the information about the model as the current value (**z**) of the parameters and a function from x_i to $\{y(x_i; \mathbf{z}), \nabla_{\mathbf{a}=\mathbf{z}} y(x_i; \mathbf{a})\}$, where the elements of the gradient vector $\nabla_{\mathbf{a}=\mathbf{z}} y$ are $\partial y(x_i; \mathbf{a})/\partial a_j$ evaluated at $\mathbf{a} = \mathbf{z}$. We express the range of this function in terms of a TaylorCoefficient1d<T> object that combines a value and an array of gradients:

AutoDeriv/TaylorCoefficient1d.h

```
template<class T>
class TaylorCoefficient1d :
    public virtual Array1d<T>,    // Gradients
    private FormedArray1d<T> {
public:
    TaylorCoefficient1d(const T& value, Subscript n_grads);
    TaylorCoefficient1d(const T& value, const ConstArray1d<T>& gradients);
```

```
    const T& value() const { return the_value; }
    T&       value()       { return the_value; }

    // Access declarations and assignment operators ... not shown
private:
    T the_value;
};
```

The function value $y(x_i; \mathbf{z})$ is stored in the_value and the gradients $\nabla_{\mathbf{a}=\mathbf{z}} y(x_i; \mathbf{a})$ are stored in the FormedArray1d<T> base subobject. The constructors (not shown) fill the base subobject and value member datum from constructor arguments.

The DataModel<T> interface acts like a function from x_i to the Taylor coefficients:

DataModeling/DataModel.h

```
template<class T>
class DataModel :
    public virtual Functional< T, TaylorCoefficient1d<T> > {
public:
    virtual       Array1d<T>& parameters()       = 0;
    virtual const Array1d<T>& parameters() const = 0;
};
```

The Array1d<T> access provided by the parameters() function allows **a** to be modified, regardless of how the parameters are actually stored by a particular class in the DataModel<T>.

Defining these interface classes partitions the problem: We can proceed to implement DataModel<T> and its gradients or we can use the interface and work on building and solving the linearized equations. In the next section, we turn to computing the partial derivatives. Then in Section 19.6 we use the interface to build the simultaneous equations and solve them with the SVD method using the the LAPACK example from Chapters 18 and 15.

19.4 Classes for Automatic Derivatives

We compute partial derivatives by *automatic differentiation*. Automatic derivatives [95, 13] simultaneously compute a function's value and derivatives by replacing the numbers in a function with multicomponent objects whose algebraic properties incorporate differentiation. For single derivatives, a two-component object combining a value and a derivative replaces computation with a value alone: $\{y, dy/dx\}$ replaces y in functions of y. Two-component constants have zero derivatives: constant 5 becomes $\{5, 0\}$. Two-component variables have derivative 1: a variable with value 5 becomes $\{5, 1\}$. When a function is evaluated with

Expression	$\partial/(\partial x_0)$	*Rall Rule*	*Rall Value*
x	**0**	$(f, 0.0)$	$(0.5, 0.0)$
x_0	**1**	$(f, \mathbf{1.0})$	$(0.2, 1.0)$
a	0	$(f, 0.0)$	$(0.4, 0.0)$
$x - x_0$	-1	$(f - g, f' - g')$	$(\mathbf{0.1}, -1.0)$
$(x - x_0)^2$	$2(x - x_0)(-1)$	$(f * g, f * g' + f' * g)$	$(0.01, -0.2)$
$(x - x_0)^2/a^2$	$2(x - x_0)(-1)/a^2$	$(f/c, f'/c)$	$(0.025, -0.05)$
$e^{(x-x_0)^2/a^2}$	$e^{(x-x_0)^2/a^2}2(x - x_0)(-1)/a^2$	$(e^f, f'e^f)$	$(1.025, -0.05127)$

Table 19.1 Example of Automatic Differentiation. The first three rows define Rall numbers. Evaluation of $e^{(x-x_0)^2/a^2}$ and its derivative with respect to x_0 begins on the fourth line and proceeds down the table. The values in the last column result from each step of the expression computation; the analytic formula for the derivative in the second column is not coded.

a two-component object, like $f(\{5, 1\})$, the numerical computations give a two-component result, the value of the function and the value of its derivative at the same point, like $\{f(5), (df/dy)(5)\}$.

To see how this works, consider a two-component object with a symbol for its value x, $\{x, 1\}$, representing a variable. The rule relating functions of these objects to functions of the components is

$$f\left(\{y, \frac{dy}{dx}\}\right) = \{f(y), \left(\frac{df}{dy}\right) * \left(\frac{dy}{dx}\right)\}. \tag{19.6}$$

It embodies the chain rule for differentiation. For example, the function $\sin(x^2)$ will start with $\sin(\{x, 1\}^2)$. Squaring $\{x, 1\}$ gives $\{x^2, 2x\}$, by the chain rule, and taking the sine gives $\{\sin(x^2), 2x\cos(x^2)\}$.

For numerical work, the computations are not done symbolically, but the values of the function and its derivatives are computed on the fly: Starting with $\{0.5, 1\}$ in $\sin(x^2)$ gives $\sin(\{0.25, 1\})$ and the sine of this result is $\{\sin(0.25), \cos(0.25)\}$. Table 19.1 works through another example, the evaluation of a Gaussian function and its derivative, step by step.

The method can be applied to any machine-computable differentiable function and, once the rules for simple functions are coded, any function built up from those functions can be differentiated without the possible errors in algebra or transcription that plague analytic differentiation and without the problems of step size that plague numerical differentiation. On the down side, the computation of analytic derivatives can be hand tuned for efficiency. Occasionally the difference will be significant, but often the complexity of analytic differentiation is not rewarded.

To use automatic derivatives, we create the multicomponent objects, code their algebraic properties, and then define functions to accept these objects. We shall call the multicomponent objects *Rall numbers* after L. B. Rall, the mathematician who developed the automatic differentiation technique for computers [95]. Specifically, we develop class `Rall1d<T,V>` for multiple-parameter first derivatives. Multiple-parameter objects have vectors of partial derivatives (higher-order derivatives add higher-dimensional components). We parameterize the class by both the value type `T` and by the gradient type `V` to allow the gradient vector to be, for example, rigid, formed, or elastic (see page 593) or even allow it to be composed of Rall numbers itself (see Exercise 19.3). For concreteness when reading the code, think of `T` as, say, `float` and `V` as, say, `RigidArithmetic1d<double, 6>` from page 503.

Most of the functions we need for `Rall1d<T, V>` can be obtained from the algebraic structure categories:

AutoDeriv/Rall1d.h

```
template<class T, class V> // T element type, V gradient vector type.
class Rall1d :
    public DivisionAlgebraCategory< Rall1d<T,V> ,T>,
    public EquivalentCategory< Rall1d<T,V> >,
    public MetricSpaceCategory< Rall1d<T,V>, Rall1d<T,V> > {
public:
    typedef T EltT;

    Rall1d(){}                                          // Constant, zero.
    Rall1d(const T& c, Subscript nvars = 1);            // Constant.
    Rall1d(const T& t, Subscript i_th, Subscript nvars); // i_th unit vector, value t.
    Rall1d(const Rall1d<T, V >& r);
    Rall1d(const T& tn, const V& vn);
    Rall1d<T,V>& operator=(const T& c);                 // Change to constant, value c.
    Rall1d<T,V>& operator=(const Rall1d<T,V>& r);
```

The metric space category endows the class with the ability to compute norms; see Exercise 19.2. We need constructors for both constants and variables as declared earlier.

The categories require declarations of many user-must-define functions, to support the various numerical operations we may find in data model expressions:

AutoDeriv/Rall1d.h

```
    // Algebra interface
Rall1d<T,V>& operator*=(const Rall1d<T,V>& rhs);
Rall1d<T,V>& setToOne();
Rall1d<T,V>& operator/=(const Rall1d<T,V>& rhs);
```

```
Rall1d<T,V>& operator+=(const Rall1d<T,V>& rhs);
Rall1d<T,V>& setToZero();
Rall1d<T,V>& operator-=(const Rall1d<T,V>& rhs);

Rall1d<T,V>& operator*=(const T& s);
Rall1d<T,V>& operator/=(const T& s);

Rall1d<T,V> dot(const Rall1d<T,V>& rhs) const { return *this * rhs; }

Boolean equivalentTo(const Rall1d<T,V>& rhs) const;

// Transcendental Functions
friend Rall1d<T,V> exp(const Rall1d<T,V>&);
friend Rall1d<T,V> sqrt(const Rall1d<T,V>&);
friend Rall1d<T,V> log(const Rall1d<T,V>&);
friend Rall1d<T,V> log10(const Rall1d<T,V>&);
friend Rall1d<T,V> cos(const Rall1d<T,V>&);
friend Rall1d<T,V> sin(const Rall1d<T,V>&);
friend Rall1d<T,V> acos(const Rall1d<T,V>&);
friend Rall1d<T,V> asin(const Rall1d<T,V>&);
```

Finally we complete the class definition for Rall1d<T, V> by defining the data-member access functions and the members themselves:

AutoDeriv/Rall1d.h

```
    T&       value()       { return t; }
    const T& value()  const { return t; }
    V&       gradient()    { return grad; }
    const V& gradient() const { return grad; }

    friend ostream& operator<<(ostream& os, const Rall1d<T,V>& r);
protected:
    T t;
    V grad;
    void make_unit(Subscript i_th);
};
```

Despite the many functions declared here, there are only constructors and arithmetic functions. We show one of each. The unit vector constructors set 1.0 into the gradient at the appropriate location:

AutoDeriv/Rall1d.c

```
template<class T, class V>
Rall1d<T,V>::Rall1d(const T& v, Subscript i_th, Subscript nvars) :
    t(v), grad(nvars) {
    make_unit(i_th);
}
```

```
template<class T, class V>
void Rall1d<T,V>::make_unit(Subscript i_th) {
    if (i_th < grad.shape(0)) {
        grad = T(0.0);    // T() forces op=(const T&), not copy of V(Subscript);
        grad(i_th) = 1.0;
    }
    else throw RallInvalidUnitErr();
}
```

Notice that we explicitly construct a scalar from the constant zero in T(0.) then assign it to the grad variable. This prevents conversion of the constant zero to a temporary of type V, as might occur if V has a constructor to set its size from an integer argument. The trivial exception class RallInvalidUnitErr is not shown.

Inside each of the user-must-define functions the elemental automatic differentiation code is written. Here, for example, is division:

AutoDeriv/Rall1d.c

```
template<class T, class V>
Rall1d<T,V> & Rall1d<T,V>::
operator/=(const Rall1d<T,V>& rhs) {
    T rhs_val = rhs.value();
    if (rhs_val == T(0.)) throw RallDivideZeroErr();
    grad  = (grad * rhs_val - value() * rhs.grad) / sqr(rhs_val);
    t     /= rhs_val;
    return *this;
}
```

The class template Rall1d<T, V> represents a category of classes for automatic differentiation to first order in multiple parameters. A specific class in the category is created by instantiating the template with a scalar and vector type. We can fix just the vector type by deriving a class template parameterized by the scalar type from the general template. For example, we can fix the vectors to be RigidArithmetic1d<T, nvars> from page 503 by deriving RigidRall1d<T, nvars> from Rall1d<T, V>:

AutoDeriv/RigidRall1d.h

```
template<class T, Subscript nvars>
class RigidRall1d :
    public Rall1d<T, RigidArithmetic1d<T,nvars> > {
public:
        // constants
    RigidRall1d() :              Rall1d<T, RigidArithmetic1d<T,nvars> >(0.0, nvars) {}
    RigidRall1d(const T& c) : Rall1d<T, RigidArithmetic1d<T,nvars> >(c, nvars)   {}
        // variables
    RigidRall1d(const T& v, int i_th, int n = nvars) :
        Rall1d<T, RigidArithmetic1d<T,nvars> >(v, i_th, nvars) {
    }
```

```
    RigidRall1d(const Rall1d<T, RigidArithmetic1d<T, nvars> >& r):
        Rall1d<T, RigidArithmetic1d<T, nvars> >(r) {
    }
};
```

We can try these Rall numbers on a test case given by Rall [94]:

ch19/tRall.C

```
typedef RigidRall1d<double, 1> RallT;
RallT x(3.0, 0, 1);
cout << "(x^2 + 2x - 3)/(x + 2) at x=3.0 is "
     << ( x * x + 2.0 * x - 3.0 ) / ( x + 2.0 )
     << endl;
```

The expression is evaluated using the Rall number x as a variable with a one-element gradient set to 1 and a value of 3. Running the code produces

```
(x^2 + 2x - 3)/(x + 2) at x=3.0 is (2.4,[1.12])
```

for the expression and derivative values at the point $x = 3$.

19.5 Example: Iterative Solution of Nonlinear Equations

Our peak-fitting problem uses derivatives to set up and solve nonlinear least squares by an algorithm called damped SVD. The setup and solution has many steps. As an aid to understanding and testing our code, we first tackle a simpler nonlinear problem, the simultaneous solution of nonlinear equations by the Newton-Raphson algorithm. We worked on this problem and the least squares problem at the same time so that we could share a common solution of nonlinear equations.

For our example, we solve the simultaneous nonlinear equations used as an example by Gorlen et al. [51, page 94], their Eq. (5.3):

$$\begin{aligned} f_0(a_0, a_1, a_2) &= 16a_0^4 + 16a_1^4 + a_2^4 - 16 \\ f_1(a_0, a_1, a_2) &= a_0^2 + a_1^2 + a_2^2 - 3 \\ f_2(a_0, a_1, a_2) &= a_0^3 - a_1 \end{aligned} \tag{19.7}$$

The Newton-Raphson algorithm for solving these equations places the three unknowns in a vector $\mathbf{a} = (a_0, a_1, a_2)$ and the three equations in a vector of functions, $\mathbf{f} = (f_0, f_1, f_2)$. Then each equation can be expanded in a first-order Taylor series:

$$f_i(\mathbf{a} + \boldsymbol{\delta}\mathbf{a}) \approx f_i(\mathbf{a}) + \sum_{j=0}^{2} \frac{\partial f_i}{\partial a_j} \delta a_j. \tag{19.8}$$

These equations are set to zero simultaneously to give linear equations for the corrections $\delta\mathbf{a}$. For $\mathrm{x} = \delta\mathbf{a}$ we must solve $\mathbf{A}\mathrm{x} = \mathrm{b}$, where

$$A_{ij} = \frac{\partial f_i}{\partial a_j} \tag{19.9}$$

$$b_i = -f_i(\mathbf{a}_{\text{trial}}). \tag{19.10}$$

The corrections $\delta\mathbf{a}$ are added to the current solution vector and the process is repeated until it converges:

$$\mathbf{a}_{\text{new}} = \mathbf{a}_{\text{trial}} + \delta\mathbf{a}. \tag{19.11}$$

Newton-Raphson solution of simultaneous nonlinear equations shares some similar features with all nonlinear problems. For example, Eq. (19.8) is similar to Eq. (19.3). As we developed our model-optimization code, we worked to unify the nonlinear optimization components in both the model-optimization and the simultaneous nonlinear equations cases. The result was an abstraction for nonlinear optimization usable for other problems. We describe this abstraction first before specializing to the Newton-Raphson example.

19.5.1 Iteration of Linearized Nonlinear Equations

Nonlinear optimization iterates toward a solution by successive approximations; the iteration process has state: the number of iterations, the current solution, and its proximity to a convergent point. Formulating the iteration as an object combines the activities of iteration with the state.

Observing our early designs, we noticed that nonlinear iterations for different equations—Newton-Raphson, Levenberg-Marquardt fitting [93, Section 15.5], or the damped SVD method we shall use later—share a common process and state and vary only in the details of the equations. This led us to introduce a common interface for the equations:

DataModeling/IteratedEquations.h

```
template<class T>
class IteratedEquations {
public:
    virtual T       update(Array1d<T>& a)       = 0;   // Returns norm of update vector.
    virtual Boolean converged()           const = 0;
    virtual         ~IteratedEquations()        {}
};
```

Each call to update() solves for $\delta\mathbf{a}$ and applies Eq. (19.11). Some equations that can be iterated are derived from this interface, as shown in Fig. 19.1.

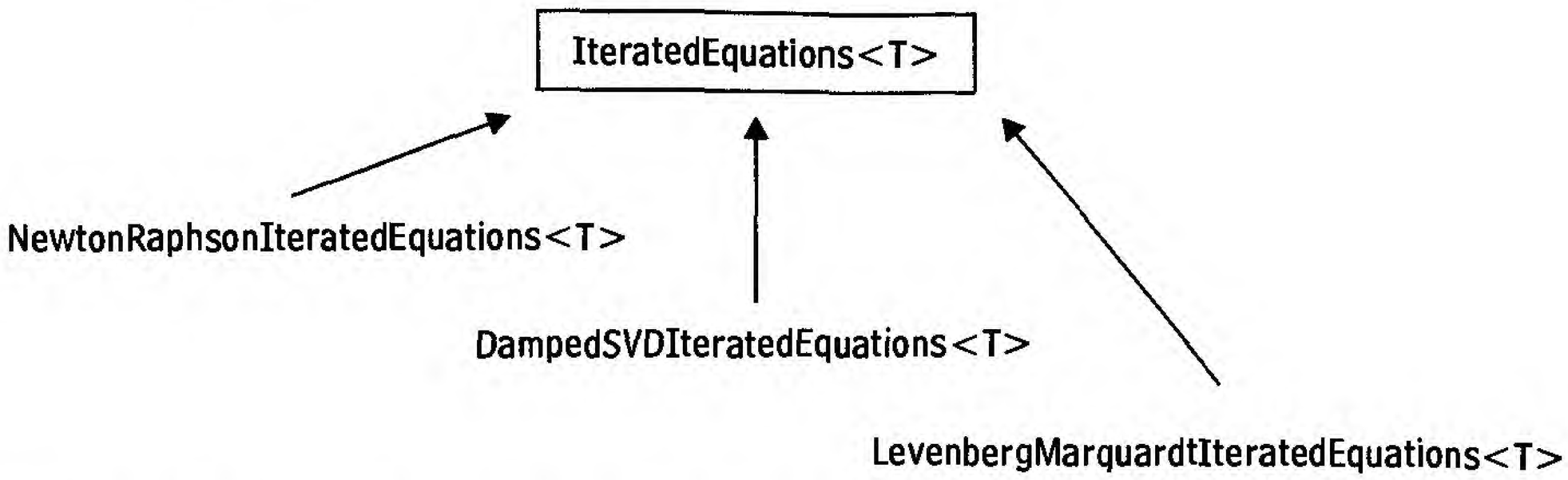


Figure 19.1 Some Classes in the IteratedEquations<T> Category. The class LevenbergMarquardtIteratedEquations<T> is not described in the text.

The linear iteration of these equations is represented by an iterator over Array1d<T> objects, with the value of each iteration being **a**, the current parameter set:

DataModeling/LinearizationIterator.h

```
template<class T>
class LinearizationIterator {
public:
    LinearizationIterator(
        IteratedEquations<T>& eqns,
        const Array1d<T>& initial_values,
        int maximum_iterations,
        T update_convergence = default_correction_convergence
);
Boolean                 more()    const;
void                    advance();
const Array1d<T>& current() const { return cur; }

void solve();

static T default_correction_convergence;
friend ostream& operator<<(ostream& os, const LinearizationIterator<T>& l);
protected:
    IteratedEquations<T>& the_eqns;
    FormedArray1d<T> cur;

    int the_max_iters;
    int the_iters;

    T correction_norm;
    T correction_convergence;    // Tested for no movement towards root.
};
```

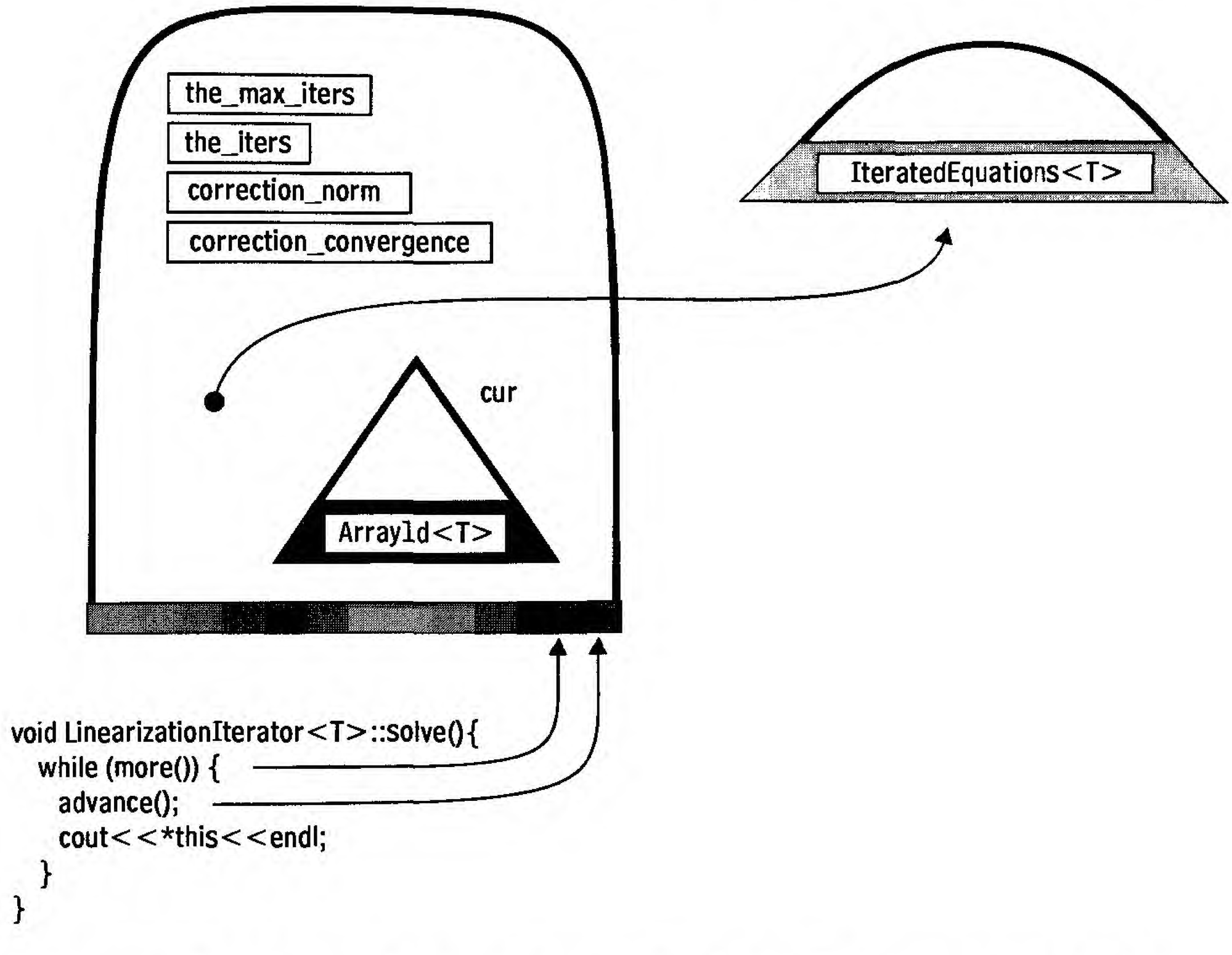


Figure 19.2 Sketch for a LinearizationIterator<T> Object. This iterator contains (at least) some object to hold the current parameter set (cur) and a reference to an IteratedEquations<T> object. Iteratorlike calls, including those from the solve() member function shown here, see only an iterator object.

The iterator object holds a reference to the IteratedEquations<T> interface, which parallels our usual Iterator formulation (cf. Section 13.7). A sketch of these objects is shown in Fig. 19.2.

The solve() member of LinearizationIterator runs the iterator object until it converges, printing the state of the iteration as it proceeds:

DataModeling/LinearizationIterator.c

```
template<class T>
void LinearizationIterator<T>::solve() {
    while (more()) {
        advance();
        cout << *this << endl;
    }
}
```

The output operator reports the state of the iteration:

DataModeling/LinearizationIterator.c

```
template<class T>
ostream& operator<<(ostream& os, const LinearizationIterator<T>& l) {
    os << "Iteration: " << l.the_iters <<
        " correction norm: " << setw(10) << setprecision(3) << l.correction_norm <<
        " Parameters:" << l.current();
    return os;
}
```

The more() and advance() functions connect the LinearizationIterator<T> and its IteratedEquations<T>. The iterator operates on any object in a class derived from the IteratedEquations<T> interface base class. The classes in this category provide update(), needed by advance():

DataModeling/LinearizationIterator.c

```
template<class T>
void LinearizationIterator<T>::advance(){
    correction_norm = the_eqns.update(cur);
    the_iters++;
}
```

and converged(), needed by more():

DataModeling/LinearizationIterator.c

```
template<class T>
Boolean LinearizationIterator<T>::more() const {
    if (the_iters == 0) return Boolean::true;    // no info yet.
    return
        the_iters < the_max_iters &&
        !the_eqns.converged() &&
        correction_norm >= correction_convergence;
}
```

Given any IteratedEquations<T> object, we can solve it with our LinearizationIterator<T>. The numerical work is done in the IteratedEquations<T> update() function. This function accepts the current values as any object in ArrayId<T>, allowing for different storage implementations and for different object types T in the array.

19.5.2 Newton-Raphson Iterations

For Newton-Raphson iterations, the current values are our trial vector $\mathbf{a}_{\text{trial}}$ in Eq. (19.11). To compute the update, we need to have the simultaneous equations, $\mathbf{f} = \{f_0, f_1, f_2\}$, and an implementation of Eqs. (19.8)–(19.11). We represent each of the equations f_i by an instance of the following class:

DataModeling/NewtonRaphsonIteratedEquations.h

```
template<class RallT>
class NewtonRaphsonFunction {
```

```
public:
    typedef RallT (*RallFunctionType)(const Array1d<RallT>& a);      // Type for f_i(a)

        // Construct from built-in function
    NewtonRaphsonFunction(RallFunctionType fp = 0) : f_i(fp) {}

        // Evaluate f_i(a)
    TaylorCoefficient1d<RallT::EltT> operator()(const Array1d<RallT>& a) const {
        return reform(f_i(a));
        }
private:
    RallFunctionType f_i;
};
```

This class sets the type for functions usable in our Newton-Raphson iterations: They are functions taking arrays of **a** values as Rall numbers and returning Rall numbers. The Rall numbers give us both the function value and gradient we need for the Taylor series expansion.

The NewtonRaphsonFunction class also converts between our standard TaylorCoefficient1d form for the value-gradient pair and the particular form for this pair stored in the template parameter RallT. The conversion is performed in a global function template reform:

AutoDeriv/Rall1d.c

```
template<class T, class V>
TaylorCoefficient1d<T> reform(const Rall1d<T,V>& r) {
    Subscript n = r.gradient().numElts();
    TaylorCoefficient1d<T> tmp(r.value(), n);
    while(n--) tmp(n) = r.gradient()(n);
    return tmp;
}
```

Had we added a conversion function for either the Rall numbers to TaylorCoefficient1d<T> or vice versa, the two implementations would have been coupled, giving us less flexibility to change either. We chose decoupling through a function that performs copies over an interface to avoid slowing down the Rall number computations.

The update algorithm is embedded in NewtonRaphsonIteratedEquations<RallT>'s update() member function:

DataModeling/NewtonRaphsonIteratedEquations.h

```
template<class RallT>
class NewtonRaphsonIteratedEquations :
    public virtual IteratedEquations<RallT::EltT>,
    public virtual Array1d< NewtonRaphsonFunction<RallT> >,
    private FormedArray1d< NewtonRaphsonFunction<RallT> > {
```

```
    public:
        typedef RallT::EltT T;
        NewtonRaphsonIteratedEquations(
            int n_eqns,
            T function_convergence = NumericalLimits<T>::epsilon
        );

        virtual RallT::EltT update(Array1d<RallT::EltT>& a);
        virtual Boolean    converged() const;

        // Assignment operators ... not shown
    private:
        T function_value;
        T function_value_convergence;
};
```

This class has an `IteratedEquations<T>` interface for the `LinearizationIterator<T>` to use and an `Array1d<T>` interface with `T` equal to `NewtonRaphsonFunction<RallT>` to allow functions to be set in. The simultaneous equations are stored as an array of functions—for example, $f_0(a_0, a_1, a_2)$, $f_1(a_0, a_1, a_2)$, $f_2(a_0, a_1, a_2)$.

The `NewtonRaphsonIteratedEquations<RallT>` constructor sets up the sizes and stores the threshold for convergence of the norm of the function values:

DataModeling/NewtonRaphsonIteratedEquations.c

```
template<class RallT>
NewtonRaphsonIteratedEquations<RallT>::
NewtonRaphsonIteratedEquations(int n_eqns, T function_convergence) :
    FormedArray1d< NewtonRaphsonFunction<RallT> > (n_eqns),
    function_value(function_convergence),    // converged()==false 1st time for sure.
    function_value_convergence(function_convergence) {
}
```

The equations are set by assignment, as we will see in the example on page 602.

The `update()` member applies Eq. (19.10) to attempt to zero all the functions simultaneously:

DataModeling/NewtonRaphsonIteratedEquations.c

```
template<class RallT>
NewtonRaphsonIteratedEquations<RallT>::T NewtonRaphsonIteratedEquations<RallT>::
update(Array1d< NewtonRaphsonIteratedEquations<RallT>::T>& a) {
    Subscript n = a.shape(0);
    ConcreteBlas1d<T> x(n);
    arrayCopy(x, a);
```

```
FormedArray1d< RallT > a_trial(n);
for (int i = 0; i < n; i++) {
    a_trial(i) = RallT(a(i), i, n);  // Convert a_i's to Rall numbers
}

LapackUnfactored< BlasRectLURep<T> > jacobian(n, n);
ConcreteBlas1d<T> delta_a(n);

Array1d< NewtonRaphsonFunction<RallT> >& f = *this;

for (i = 0; i < n; i++) {    // Evaluate equations, unpack derivatives
    TaylorCoefficient1d<T> f_a_trial = f(i)(a_trial);
    jacobian.row(i) = f_a_trial;
    delta_a(i)      = f_a_trial.value();
}

jacobian.factor().solve(delta_a);     // Solve for correction.
x -= delta_a;                         // Apply correction.
arrayCopy(a, x);                      // Store results.
function_value = x.norm();            // Save for convergence test.
    return delta_a.norm();
}
```

Notice that we are using both concrete and interfaced arrays in this code. Since assigning of arrays in these two array systems is not provided, we assign one kind of array to another using the arrayCopy() function template developed in Exercise 13.14.

To apply our Newton-Raphson code to Eq. (19.7), we write the equations as function templates to allow expansion over any arithmetic type, including our Rall numbers:

ch19/tNewtonRaphson.C

```
template<class T>
T gorlen5p3_f(const Array1d<T>& a){
    return 16.0 * pow(a(0), 4) + 16.0 * pow(a(1), 4) + pow(a(2), 4) - T(16.0);
}

template<class T>
T gorlen5p3_g(const Array1d<T>& a){
    return a(0)*a(0) + a(1)*a(1) + a(2)*a(2) - T(3.0);
}
```

```
template<class T>
T gorlen5p3_h(const Array1d<T>& a){
    return pow(a(0), 3) - a(1);
}
```

When we assign them to the NewtonRaphsonIteratedEquations entries, the templates expand to the RallT type used in the code:

ch19/tNewtonRaphson.C

```
cout << "Finding Zeros of Nonlinear Equations Test " << endl
     << "Equations 5.3 in Gorlen, Orlow and Plexico, " << endl
     << "Data Abstraction and Object Oriented Programming in C++" << endl;

typedef RigidRall1d<double, 3> RallT;

NewtonRaphsonIteratedEquations<RallT> eqns(3);    // 3 equations in 3 unknowns

eqns(0) = gorlen5p3_f;    // Store functions
eqns(1) = gorlen5p3_g;
eqns(2) = gorlen5p3_h;

FormedArray1d<double> rhs(3);    // Variables
rhs = 1.0;                       // Initial guess = [1, 1, 1]

        // Solve equations, using at most 10 iterations
LinearizationIterator<double> solver(eqns, rhs, 10);
solver.solve();

cout << "Zeros are: "
          << setprecision(NumericalLimits<double>::digits) << solver.current() << endl;
```

The program produces the following output:

```
Finding Zeros of Nonlinear Equations Test
Equations 5.3 in Gorlen, Orlow and Plexico,
Data Abstraction and Object Oriented Programming in C++
Iteration: 1 correction norm:          0.361 Parameters:[0.929, 0.787, 1.28]
Iteration: 2 correction norm:           0.11 Parameters:[0.887, 0.693, 1.32]
Iteration: 3 correction norm:         0.0207 Parameters:[0.878, 0.677, 1.33]
Iteration: 4 correction norm:       0.000574 Parameters:[0.878, 0.677, 1.33]
Iteration: 5 correction norm:       4.52e-07 Parameters:[0.878, 0.677, 1.33]
Iteration: 6 correction norm:        2.8e-13 Parameters:[0.878, 0.677, 1.33]
Iteration: 7 correction norm:              0 Parameters:[0.878, 0.677, 1.33]
Zeros are: [0.877965760274298, 0.676756970517829, 1.33085541162123]
```

This demonstrates our Rall number implementation and the concept of the LinearizationIterator.

19.6 Classes for Damped SVD Equations

Having specified the data-entry interface and implemented it in a simple way (Section 19.2), and having specified a data-model interface (Section 19.3) and an interface for iterated nonlinear equations (Section 19.5.1), we turn now to setting up and solving the nonlinear equations for peak fitting. We shall create a class called DampedSVDIteratedEquations, comparable to NewtonRaphsonIteratedEquations, that updates trial parameters using damped singular value decomposition of the linear least squares equations. To understand how this update will be done, we must explain more about the damped SVD solution.

We build on the robust solver for linear equations using singular value decomposition that was developed in Section 18.7.3. Experience has shown that directly minimizing the linear least squares problem may not lead toward the minimum of the nonlinear problem due to errors of linearization. Moreover, noise in the data or linear approximation error may cause two or more of the parameters **a** to become indistinguishable (degenerate). Both problems lead to erroneously large corrections that can lead away from the minimum. To circumvent these problems, we use the *damped least squares* technique of Marquardt [76], as adapted to SVD in modern nonlinear least squares methods [3, 71].

The simplest derivation for the Marquardt technique starts with the empirical observation that solution by linear approximation of nonlinear equations can lead to parameter changes that are too large. We therefore augment the original χ^2 with a penalty for large corrections:

$$\hat{\chi}^2 = (\mathbf{Ax} - \mathbf{b})\,(\mathbf{Ax} - \mathbf{b})^T + \lambda^2 \mathbf{x}\mathbf{x}^T. \tag{19.12}$$

Since $\mathbf{x}\mathbf{x}^T$ is the square of the norm of the correction vector, the damping factor λ^2 [70] controls how much penalty large corrections incur.

The damping adds a $\lambda^2\mathbf{x}$ term to the simultaneous equations, Eq. (19.5), for the parameter updates. The new solution becomes

$$\begin{aligned} \mathbf{x} &= \mathbf{V}\boldsymbol{\Sigma}_\lambda^+ \mathbf{U}^T \mathbf{b} \\ \left(\boldsymbol{\Sigma}_\lambda^+\right)_{jj} &= \left[\left(\boldsymbol{\Sigma}^2 + \lambda^2 \mathbf{I}\right)^{-1} \boldsymbol{\Sigma}\right]_{jj} \\ &= \frac{\boldsymbol{\Sigma}_{jj}}{\boldsymbol{\Sigma}_{jj}^2 + \lambda^2}. \end{aligned} \tag{19.13}$$

With a good model that is close to linear, λ can be set small and $\mathbf{\Sigma}_\lambda^+ \to \mathbf{\Sigma}^+$ as defined in Section 18.7, giving the pseudoinverse solution to the $\mathbf{Ax} = \mathbf{b}$ problem [70]. For a poorer model or far from the minimum, λ^2 can be set large and $\mathbf{\Sigma}_\lambda^+ \to \mathbf{\Sigma}/\lambda^2$. Then $\mathbf{x} = \mathbf{A}^T\mathbf{b}/\lambda^2$, a solution known as the *steepest-descent* solution. This solution ignores error-surface shape information in favor of simply descending along all coordinates independently. Thus for $\lambda^2 \gg \mathbf{\Sigma}_{00}$ (the largest singular value) we have a sure but slow nonlinear optimization step.

Intermediate values of λ^2 compromise between these extremes. In theory [3], the pseudoinverse and steepest-descent directions tend to be nearly orthogonal, and the direction to the true minimum lies in between. Imagine being high on the hill at one end of a long, curved valley. Steepest descent heads directly down the hill into the valley floor, ignoring the minimum off in the distance. The pseudoinverse direction tends to be parallel to the valley floor. Intermediate values of λ^2 lead partly down hill, partly along the valley, which is a more direct path to the final minimum.

In practice we start off with some value for λ^2 and check if we are going downhill in nonlinear χ^2. If so, we gain confidence and reduce λ^2 to accelerate our convergence; if not, we back up and increase λ^2 to move slower but more definitely downhill. *Numerical Recipes* [93, Section 15.5] suggests beginning with $\lambda^2 = 0.001$ and changing by factors of 10.

To implement this damping procedure, we derive a DampedRectSVDRep<T> class from the RectSVDRep<T> class on page 572:

LapackWrap/DampedRectSVDRep.h

```
template<class T>
class DampedRectSVDRep :
    public RectSVDRep<T> {
public:
    class Factored :
        public RectSVDRep<T>::Factored {
    public:
        Factored(Unfactored* mp) :
            RectSVDRep<T>::Factored(mp),
            lambda2(NumericalLimits<T>::epsilon) {
        }

        T  dampingFactor() const { return lambda2; }
        T& dampingFactor()       { return lambda2; }
protected:
        virtual void pseudoInvert(ConcreteBlas2d<T>& utb);
        virtual void pseudoInvert(ConcreteBlas1d<T>& utb);
```

```
    private:
        T lambda2;
    };
};
```

The initial damping factor is set small, yielding the pseudoinverse solution; the class adds a function to set the damping factor into the factored matrix and overrides the virtual function for the pseudoInvert function:

LapackWrap/DampedRectSVDRep.c

```
template<class T>
void DampedRectSVDRep<T>::Factored::pseudoInvert(ConcreteBlas1d<T>& b) {
    for (Subscript i = 0; i < k; i++) {
        T s_i = sigma(i);
        b(i) *= s_i / ( sqr(s_i) + lambda2 );
    }
}
```

This replaces the sharp cutoff on page 573 with the damping. The other pseudoInvert() member function (not shown) works for multiple right-side vectors.

With a suitable linear equation solver in hand, we now need to construct the linear equations from the DataModel<T> (page 589) and PhysicalData<T> (page 585) interfaces. To get our bearings, let's compare where we are at this point to the simpler Newton-Raphson case. We are headed toward designing DampedSVDIteratedEquations that parallel NewtonRaphsonIteratedEquations. In the Newton-Raphson case, we already had RectLURep<T> from Chapter 18 to solve the linearized equations; we just completed the extension of RectSVDRep<T> to the damped case so that we can solve the linearized equations for damped SVD least squares.

It remains to set up the equations and apply the solution. In the Newton-Raphson case, we did both steps in the update() function, page 600. Since the least squares equations are more complex and since the formation of these equations could be used with other linear equation solvers, we factor out the equations setup into a separate class called LeastSquares.

Instances of of LeastSquares are similar to augmented matrices $(\mathbf{A} : \mathbf{b})$, a combination of $\mathbf{A}$ and $\mathbf{b}$ often used in least squares problems. However, LeastSquares itself represents only the construction of $\mathbf{A} : \mathbf{b}$, not the element storage, factoring algorithm, or memory management strategy for the matrix. Thus we parameterized the class LeastSquares by A, where A is intended to be a class similar to LapackUnfactored< RectSVDRep<double> >:

DataModeling/LeastSquares.h

```
template<class A>
class LeastSquares :
    public A {      // Will be e.g LapackUnfactored< RectSVDRep<double> >
```

```
public:
    typedef A::EltT EltT;
    typedef A::Representation::Unfactored   Unfactored;
    typedef A::Representation::Unknowns2d Unknowns2d;
    typedef A::Representation::Knowns2d    Knowns2d;
    typedef A::Representation::Unknowns1d Unknowns1d;
    typedef A::Representation::Knowns1d    Knowns1d;

    LeastSquares(
        const PhysicalData<EltT>& data,
        const Functional<EltT, TaylorCoefficient1d<EltT> >& model,
        Subscript nparms
    );

    EltT                  chi2() const { return the_chi2; }
    const Knowns1d& rhs()  const { return b;          }

    // Assignment operators ... not shown ...
private:
    ConcreteBlas1d<EltT> b;
    EltT                 the_chi2;
};
```

In addition to substituting different factoring representations in place of `RectSVD-Rep<double>`, we could also replace `LapackUnfactored<Rep>` with other matrix managers. For example, we could implement column scaling [70] by deriving from `LapackUnfactored<Rep>` and replacing the `factor()` member function with one that scales the matrix columns before factoring. This new factoring class could then be used in least squares problems by using it as the parameter `A` in `LeastSquares<A>`.

The representation class specified in the template argument `A` is accessed via the base class member function `rep()` when the equations have been set up in the constructor:

DataModeling/LeastSquares.c

```
template<class A>
LeastSquares<A>::LeastSquares(
    const PhysicalData<EltT>& data,
    const Functional<EltT, TaylorCoefficient1d<EltT> >& model,
    Subscript nparms
) :
    A(data.numElts(), nparms),
    b( data.numElts() ),
    the_chi2( EltT(0) ) {
```

```
AccessedPhysicalDataBrowser<EltT> data_b = data.browser();

ConcreteBlas2d<EltT>& a_matrix = rep();    // Connect to base class matrix.
for (Subscript i = 0; data_b.more(); i++) {
    TaylorCoefficient1d<EltT> model_i = model( data_b.coordinate() );

    EltT recip_sigma_i   = EltT(1) / data_b.sigma();
    EltT delta_i         = ( data_b.value() - model_i.value() ) * recip_sigma_i;
    b(i)                 = delta_i;
    the_chi2             += sqr(delta_i);

    arrayCopy(a_matrix.row(i), model_i);    // Load gradients into rows of matrix.
    a_matrix.row(i) *= recip_sigma_i;

    data_b.advance();
  }
}
```

For example, if A is LapackUnfactored< RectSVDRep<double> >, then rep() returns a ConcreteBlas2d<double>, the unfactored representation for RectSVDRep<double>. The loop fills **A** and **b** and computes χ^2. Thus we have our equations set up.

Finally we are ready to design the class that can iteratively form and solve least squares problems. The two interfaces to objects needed to form the equations, PhysicalData<T> for b and DataModel<T> for A, initialize a DampedSVDIterated-Equations<T> object:

DataModeling/DampedSVDIteratedEquations.h

```
template<class T>
class DampedSVDIteratedEquations :
    public virtual IteratedEquations<T> {
public:
    DampedSVDIteratedEquations(PhysicalData<T>& data_to_fit,
        DataModel<T>& model_to_use);
    virtual const Array1d<T>& initialValues() const { return z_0; }
    virtual Boolean           converged()   const { return limit > lse.chi2(); }
    virtual T                 update(Array1d<T>&);
private:
        // Abbreviations for matrix storage wrapper classes.
    typedef LapackUnfactored< DampedRectSVDRep< T > > Unfactored;
    typedef LapackFactored< DampedRectSVDRep< T > >    Factored;
```

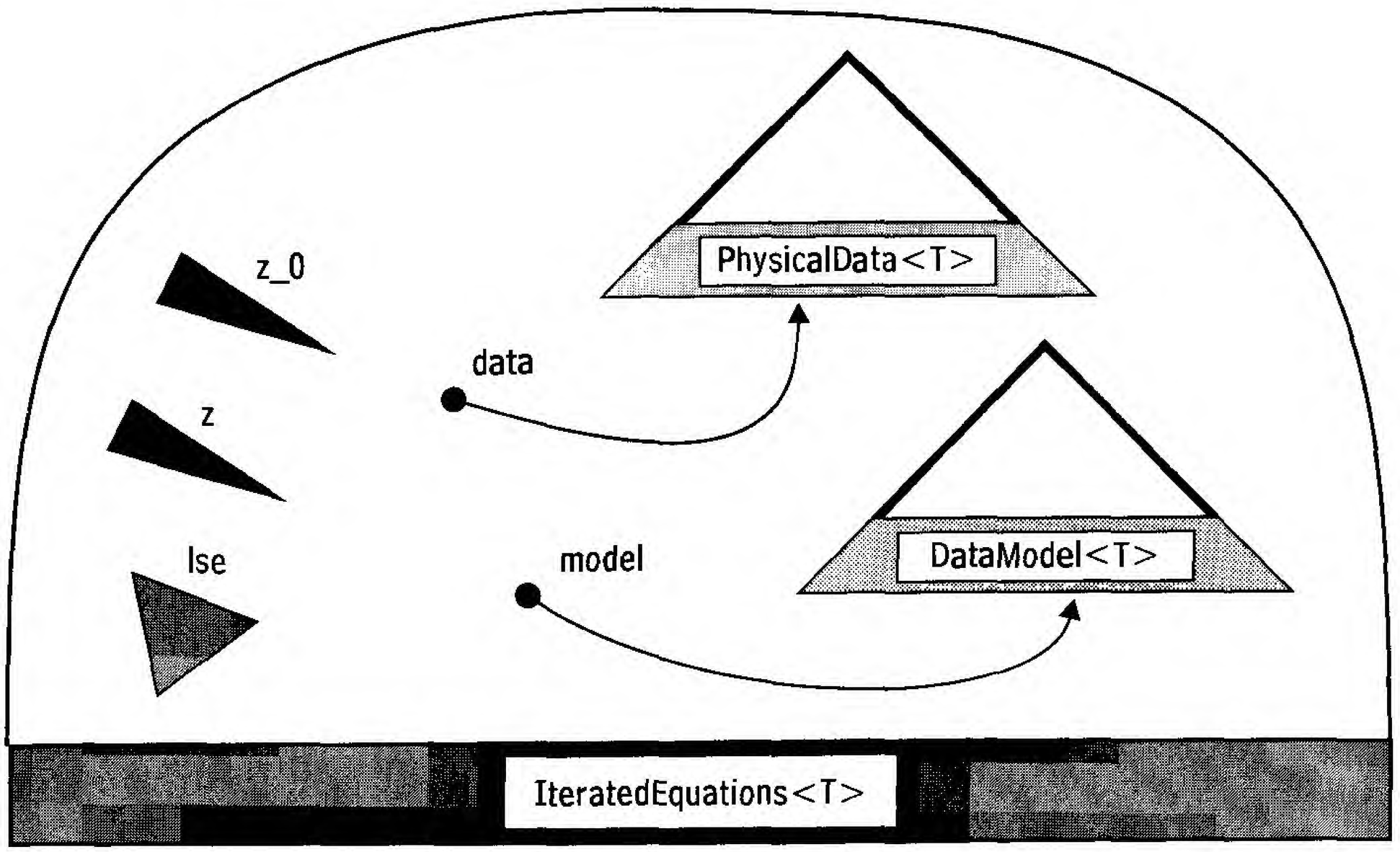


Figure 19.3 Sketch of DampedSVDIteratedEquations<T>.

```
    PhysicalData<T>&              data;       // Data
    DataModel<T>&                 model;      // Model
    FormedArray1d<T>              z_0;        // Initial parameters
    T                             limit;      // Convergence in chi^2
    T                             lambda2;    // Damping factor
    LeastSquares<Unfactored>      lse;        // Current linearization.
    ConcreteBlas1d<T>             z;          // Current parameter set.
};
```

These objects represent the nonlinear equations, with the current linearization of a DampedSVDIteratedEquations<T> stored in its LeastSquares<T> member; see Fig. 19.3.

The DampedSVDIteratedEquations<T> constructor initializes the references to the interfaces and constructs both an initial linear equation set in lse and an initial step via their solution:

DataModeling/DampedSVDIteratedEquations.c

```
template<class T>
DampedSVDIteratedEquations<T>::
DampedSVDIteratedEquations(PhysicalData<T>& data_to_fit,
        DataModel<T>& model_to_use) :
    data(data_to_fit),
    model(model_to_use),
    z_0(model.parameters()),
    lse( data, model, model.parameters().numElts() ),
    lambda2(0.001),    // Value recommended by Numerical Recipes and Marquardt.
```

```
z(model.parameters().numElts()) {
arrayCopy(z, model.parameters());
ConcreteBlas1d<T> x(z);
LeastSquares<Unfactored> trial(lse);        // Save data (from overwrite in factor).

Factored usvt = trial.factor();             // A -> U*S*V^T
usvt.rep().dampingFactor() = lambda2;
usvt.solve(x, trial.rhs());                 // x from Ax=b via damped SVD.
arrayCopy(model.parameters(), x + z);       // Initial step.
}
```

The last line feeds the first step back to the model for storage as the current model parameters.

The update() procedure contains the logic for manipulating the damping as well as the solution for a single step:

DataModeling/DampedSVDIteratedEquations.c

```
template<class T>
T DampedSVDIteratedEquations<T>::update(Array1d<T>& parms) {
        // Update model.
    model.parameters() = parms;

    // Linearize non-linear equations and evaluate new chi-squared.
LeastSquares<Unfactored> trial( data, model, parms.numElts() );

if ( trial.chi2() < lse.chi2() ) {          // If we are going down hill,
    lse = trial;                            // accept this equation set,
    arrayCopy(z, parms);                    // record its parameters,
    lambda2 /= 10.0;                        // reduce damping.
}
else {                                      // If we are going up hill,
    trial = lse;                            // overwrite this equation set,
                                            // ignore its parameters,
    lambda2 *= 10.0;                        // increase damping.
    }
    ConcreteBlas1d<T> x(parms.numElts());   // Update-vector storage.

    Factored usvt = trial.factor();         // A -> U*S*V^T
    usvt.rep().dampingFactor() = lambda2;
    usvt.solve(x, trial.rhs());             // x for Ax=b by damped SVD

    arrayCopy(parms, x + z);                // x = a - z => a = x + z.
    return x.norm();
}
```

At the top we unload the values of **a** maintained by the LinearizationIterator<T> (page 596); at the bottom we update these values. In between we build a trial linearization based on the input **a** values. If the input values give a lower χ^2 than the previous input values, the stored linearization is overwritten by the trial; if the input values are not better than our stored values, we discard the trial and redamp the stored equations. The final steps in update() solve the linear system for a new parameter set.

19.7 The main() Program

To try this, we need to bring together a model and a set of data points. We'll implement the model of Eq. (19.1) with a class in the DataModel<T> category:

ch19/DualGaussianModel.h

```
template<class T>
class DualGaussianModel :
    public virtual DataModel<T> {
public:
    DualGaussianModel(T a0, T a1, T a2, T a3, T a4, T a5);
    virtual const Array1d<T>& parameters() const;
    virtual       Array1d<T>& parameters();

    virtual TaylorCoefficient1d<T> operator()(const T& x) const;
    virtual DualGaussianModel<T>* clone() const;
private:
    Rigid Array1d<T, 6> a_values;
};
```

The current parameter values are held in the array a_values, which is initialized by the constructor (not shown). The parameters() functions required by DataModel<T> (also not shown) simply return this array. The model computation uses Rall numbers to compute simultaneously the model value and gradient:

DataModeling/DualGaussianModel.c

```
template<class T>
TaylorCoefficient1d<T> DualGaussianModel<T>::operator()(const T& x_coord) const {
    typedef RigidRall1d<T, 6> RallT;
    RallT x(x_coord);
    RigidArray1d<RallT, 6> a;
    for (Subscript i = 0; i < 6; i++) {
        a(i) = RallT(a_values(i), i, 6);
    }
```

```
    return reform(
        a(0) * exp( -sqr( (x - a(1)) / a(2) ) ) +
        a(3) * exp( -sqr( (x - a(4)) / a(5) ) )
    );
}
```

A main() function combines this model class with data points read from a file (written by another program, not shown) by FormedPhysicalData<T>. The equations are constructed by DampedSVDIteratedEquations<T> and solved with the appropriate linearization iterator, using either default iteration criteria or ones read as command line arguments:

ch19/tDampedSVD.C

```
int main(int argc, char* argv[]) {

    int       max_iters    = 10;
    double convergence = NumericalLimits<float>::epsilon;

    if (argc > 1) max_iters = String(argv[1]).strtol();
    if (argc > 2) convergence = String(argv[2]).strtod();

    // A problem from "Numerical Recipes Example Book",
    // W. Vetterling, S. Teukolsky, W. Press, and B. Flannery,
    // Cambridge University Press, 1988.

    DualGaussianModel<double> model(4.5, 2.2, 2.8, 2.5, 4.9, 2.8);
    FormedPhysicalData data(cin, 100);
    DampedSVDIteratedEquations<double> eqns(data, model);

    LinearizationIterator<double> solver(eqns, model.parameters(), max_iters,
        convergence);
    solver.solve();

    cout << solver.current() << endl;
    return EXIT_SUCCESS;
}
```

Figure 19.4 shows the result of running this code on simulated experimental data.

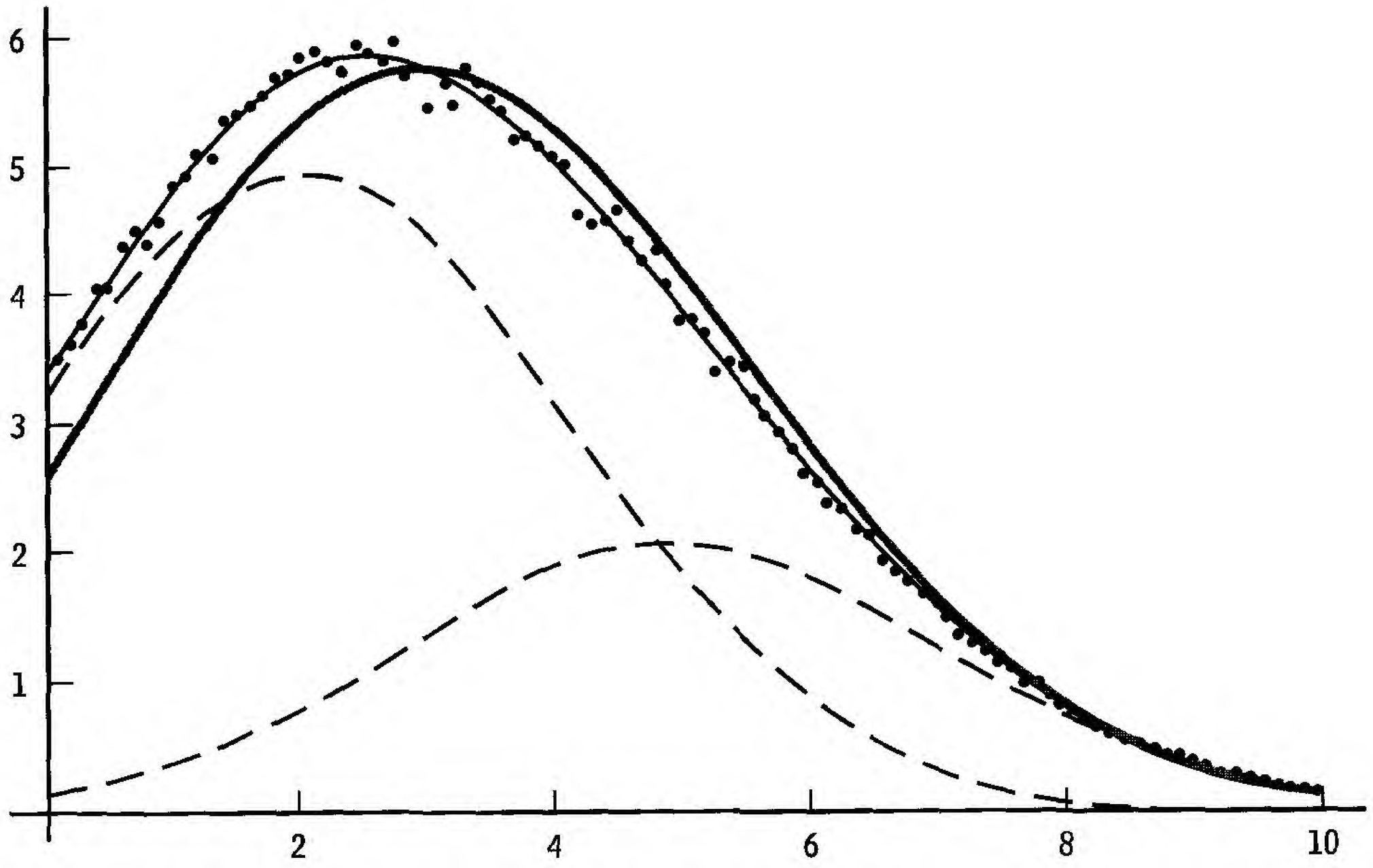


Figure 19.4 An Example of a Data Model Result. The dots present simulated experimental data, computer generated for this example by adding random numbers proportional to the signal to the sum of two Gaussian peaks. The gray curve gives one initial guess for a two-peak fit; the black curve shows the fit using the code from this chapter. The dashed curves are the separate peaks resulting from the fit.

19.8 Summary

The example in this chapter illustrated many of the classes and techniques we developed in this book. We used simple classes, interfaced classes, implementation inheritance, class templates, function-structure categories, wrapped FORTRAN code, and so on. To illustrate the full potential of C++ we deliberately applied more abstraction at each point than a typical C++ programmer might apply in the first version of a system. In other ways, we expect the work in this chapter to be more typical of scientific and engineering programming in C++ than work in earlier chapters. Notice that the work is a mix of newly designed class subsystems and extensions or applications of class systems from earlier problems. On the one hand we cannot spend all of our effort building new array classes for each new problem, but on the other hand each new problem will likely require new kinds of classes to be designed.

To be practical for any particular data modeling problem, additional work on the example code from this chapter would be required. Input errors must be trapped, nonconvergence strategies must be added, a flexible user interface must

be implemented, and more choices for input data must be accommodated. We hope that we have provided sufficient hints throughout this book for you to accomplish these extensions.

19.9 Notes and Comments

19.1 You might have noticed that we have said little about algorithms or data structures, the core of most scientific and engineering programs. Our entire focus has been on relationships and organizational structures in C++ programs and their verification by type checking. As Part I illustrates, the C++ facilities for number crunching and storage are similar to the facilities in FORTRAN and C. The strength of C++ lies in new kinds of features that assist in formulating more complex programs. Most of the work in formulating class declarations and function declarations would not be required explicitly in simpler languages; it is just this work that builds the framework for more sophisticated and reliable programs.

19.2 Our nonlinear fitting can be derived in various other ways. The *stochastic inverse* solution [3, Section 12.3.5] elegantly produces the result in a more general context that allows for noise and model parameter correlations. Since nonlinearity introduces correlations, the stochastic inverse seems like an underappreciated avenue for further improvements. The stochastic inverse also generalizes the Wiener or optimal least squares smoothing filter to non-Fourier applications. Consequently λ^2 can be interpreted as the ratio of the average noise power to the expected deviation of the parameters, squared. Usually, estimating the latter quantity is so difficult that guessing λ^2 is preferred.

19.3 A fitting algorithm similar to the one we use here, called [93, Section 15.5] *Levenberg-Marquardt*, forms *normal equations* from $\mathbf{Ax} = \mathbf{b}$ by multiplying both sides by $\mathbf{A}^T$. The resulting equations can be solved with LU decomposition (Chapter 15), but the normal equations are often poorly conditioned [70]. SVD produces reasonable answers even when the matrix in the least squares problem is nearly or exactly singular, as may happen in nonlinear problems with nearly similar parameters.

19.4 The Marquardt technique uses only first derivatives, but it is sometimes described as a method based on the curvature of the error surface, with second derivatives ignored. A true second derivative method could be based on automatic differentiation, with eigenvalues of the curvature matrix used to determine whether we are near a minimum or maximum in each parameter. In principle, this would converge quickly for the parameters we know are near minima and step away from the others by steepest descents. The extra cost of computing the correct curvature matrix and applying this principle is often defeated by the slow convergence of any parameters we use steepest descents on.

19.5 Use of the automatic derivatives here was inspired by the pioneering C++ work of Gorlen et al. [51]. See also [62].

19.10 Exercises

19.1 Implement AccessedPhysicalDataBrowser<T>.

19.2 A linear space (cf. Section 16.1) having an inner product between two vectors can define a size or *norm* for vectors and an angle between two vectors. Design and implement a MetricSpaceCategory<V, M> function-structure category for metric spaces. It should assume a user-must-define member function, dot(), that takes a vector of type V as an argument and returns the inner product of the vector for which it is called and the argument; the inner product is of type M. The category should provide a norm() member function and a global dot() function that takes two vector arguments. Should MetricSpaceCategory<V, M> be derived from LinearSpaceCategory<V, S>? Explain.

19.3 Develop a class RigidRall2d implementing second derivatives with multiple parameters. *Hint:* Use Rall1d with parameters containing Rall numbers.

19.4 Restructure the Rall1d class without constructors for constants, using only external scalar multiplication rules. Can you succeed?

19.5 Interval arithmetic computes the size of error bars due to numerical or physical uncertainties by combining the error bars for the components in an expression according to the rules for error propagation. Using these rules, develop an interval arithmetic class and test it.

19.6 The SVD vectors in $\mathbf{V}\mathbf{\Sigma}^{-1}$ are the principal axes of the confidence region for the vector $\mathbf{a} - \mathbf{z}$, assuming normally distributed errors [93, Section 15.6]. Alter the design of the classes described here to implement a way of examining the confidence region for each parameter.

19.7 The maximum likelihood for a model parameters set **a** occurs when

$$\Omega(\mathbf{a}) = \sum_{i=1}^{N} \rho\left(\frac{y_i - y(x_i; \mathbf{a})}{\sigma_i}\right) \tag{19.14}$$

is minimized over **a** [93, Section 15.7]. Here $\rho(y)$ is the negative logarithm of data error distribution function. For example, $\rho(y)$ would be y^2 if the error distribution is Gaussian ($\exp(-y^2)$). Statistics give no leeway in our choice of ρ: We are obligated to determine the distribution of our measurement errors and use the corresponding ρ function. However, optimization of such a formula may be impossible or too costly with current methods. Extend the parameter estimation program here to use Cauchy or Lorentzian distributions:

$$\rho(z) = \log\left(1 + \frac{z^2}{2}\right) \tag{19.15}$$

by altering the σ weights within a least squares model. These variances become implicit functions of the model—we give less weight to a data point far from our

estimate—and hence we can only discriminate against outliers once we are close to a converged model. Write the code to start with a Gaussian distribution and then introduce outlier rejection as the minimum is approached.

19.8 A so-called *robust estimator* (cf. [93, Section 15.7]) is an estimator that is somewhat insensitive to small deviations from the assumptions under which it was formulated. A common robust estimator uses least absolute value rather than least squares:

$$\rho z = |z| \tag{19.16}$$

Taylor expansion methods cannot apply here: Near the minimum the gradient does not approach zero. For two parameters, the error surface is conelike near its minimum. If we run down the error surface by large steepest-descent steps until we find a point on the other side of the cone (gradient of opposite sign), we can compute the intersection of the gradients to estimate the minimum. Write a program to implement this strategy and test it.

REFERENCES

1. Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA, 1985.

2. James L. Adcock. Making a list and checking it twice. *The C++ Report*, 5(3):54–57, March 1993.

3. Keiiti Aki and Paul G. Richards. *Quantitative Seismology: Theory and Methods*, volume II. W. H. Freeman and Co., San Francisco, 1980.

4. American National Standards Institute X3J11 Language Subcommittee. *American National Standard X3.159-1989*.

5. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1992.

6. Robert S. Arnold, editor. *Software Reengineering*. IEEE Computer Society Press, Los Alamitos, CA, 1993.

7. Martin C. Atkins. *Implementation Techniques for Object-Oriented Systems*. PhD thesis, Department of Computer Science, University of York, Heslington, York, YO1 5DD, England, June 1989.

8. Martin C. Atkins and Lee R. Nackman. The active deallocation of objects in object-oriented systems. *Software—Practice and Experience*, 18(11):1073–1089, November 1988.

9. John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of ACM*, 21(8):613–641, August 1978.

10. John J. Barton and Lee R. Nackman. Scientific and engineering C++: Multidimensional arrays. *C++ Report*, pages 42–45, 47, November 1993.

11. Klaus-Jürgen Bathe. *Finite Element Procedures in Engineering Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1982.

12. G. D. Bergland. A guided tour of program design methodologies. *IEEE Computer*, 14(10):13–37, October 1981. Reprinted in [123].

13. Christian Bischof, Andreas Griewank, and David Juedes. *Exploiting Parallelism in Automatic Differentiation*, volume 59, pages 146–153. ACM, June 1991. Conference in Cologne, Germany.

14. Grady Booch. Object-oriented development. *IEEE Trans. Software Engineering*, SE-12(2):211–221, February 1986.

15. Grady Booch. *Software Engineering with ADA*. Benjamin Cummings, Menlo Park, CA, second edition, 1987.

16. Grady Booch. *Object Oriented Design with Applications*. Benjamin Cummings, Redwood City, CA, 1991.

17. Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA, 1975.

18. Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, pages 10–19, April 1987.

19. James A. Brown, Sandra Pakin, and Raymond P. Polivka. *APL2 at a Glance*. Prentice Hall, Englewood Cliffs, NJ, 1988.

20. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17:471–522, 1985.

21. T. A. Cargill. Does C++ really need multiple inheritance? In *USENIX C++ Conference Proceedings*, pages 315–323. USENIX Association, April 1990.

22. Tom Cargill. *C++ Programming Style*. Addison-Wesley, Reading, MA, 1992.

23. Tsun S. Chow, editor. *Tutorial: Software Quality Assurance—A Practical Approach*. IEEE Computer Society Press, Silver Spring, MD, 1985.

24. Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

25. T. F. Coleman and C. F. Van Loan. *Handbook for Matrix Computations*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1988.

26. Martin Colloms. *Computer Controlled Testing and Instrumentation: An Introduction to the IEC-625: IEEE-488 Bus*. Pentech Press, London, 1983.

27. James O. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, Reading, MA, 1992.

28. R. Courant and D. Hilbert. *Methods of Mathematical Physics*, volume I. Interscience Publishers, New York, 1953.

29. Brad Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, MA, 1986.

30. H. S. M. Coxeter. *Introduction to Geometry*. John Wiley and Sons, New York, second edition, 1969.

31. Antonio R. Damasio and Hanna Damasio. Brain and language. *Scientific American*, 267(3):88–95, September 1992.

32. David Detlefs. Garbage collection and run-time typing as a C++ library. In *USENIX C++ Conference Proceedings*, pages 37–56. USENIX Association, August 1992.

33. David L. Detlefs. Concurrent garbage collection for C++. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 101–134. The MIT Press, Cambridge, MA, 1991.

34. Gouri Dhatt and Gilbert Touzot. *The Finite Element Method Displayed*. John Wiley and Sons, Chichester, UK, 1984.

35. Walter C. Dietrich, Jr., Lee R. Nackman, and Franklin Gracer. Saving a legacy with objects. In Norman Meyrowitz, editor, *Proceedings Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'89)*, pages 77–83. Association for Computing Machinery, October 1989. Reprinted in [6].

36. Walter C. Dietrich, Jr., Lee R. Nackman, Christine J. Sundaresan, and Franklin Gracer. TGMS: An object-oriented system for programming geometry. *Software—Practice and Experience*, 19(10):979–1013, October 1989.

37. E. Dijkstra. Programming considered as a human activity. In *Proceedings of the 1965 IFIP Congress*, pages 213–217, Amsterdam, North-Holland, 1965. Reprinted in [122].

38. E. W. Dijkstra. Structured programming. In J. N. Buxton and B. Randell, editors, *Software Engineering Techniques*, pages 84–88. NATO Scientific Affairs Division, Brussels, 1970. Reprinted in [122].

39. E. W. Dijkstra. Notes on structured programming. In *Structured Programming*, A.P.I.C. Studies in Data Processing, No. 8, chapter I, pages 1–82. Academic Press, New York, 1972.

40. J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1979.

41. Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

42. Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

43. Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. In *USENIX C++ Conference Proceedings*, pages 1–19. USENIX Association, August 1992.

44. Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

45. A. D. Falkoff and K. E. Iverson. The design of APL. *IBM Journal of Research and Development*, 17(4):324–334, July 1973.

46. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP'93: European Conference on Object-Oriented Programming*, July 1993.

47. Philippe Gautron, Kim Knuttila, and Alan D. Sloane. A survey paper: Various proposals to revise templates specifications. Technical Report X3J16/92-0014, ANSI Working Document, March 1992.

48. Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley and Sons, New York, 1982.

49. Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

50. Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, second edition, 1989.

51. Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley and Sons, Chichester, UK, 1990.

52. Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, MA, 1989.

53. A. Nico Habermann and Dewayne E. Perry. *Ada for Experienced Programmers*. Addison-Wesley, Reading, MA, 1983.

54. Daniel C. Halbert and Patrick D. O'Brien. Using types and inheritance in object-oriented programming. *IEEE Software*, 4(5):71–79, September 1987.

55. Charles A. Hall and Thomas A. Porsching. *Numerical Analysis of Partial Differential Equations*. Prentice Hall, Englewood Cliffs, NJ, 1990.

56. David Halliday and Robert Resnick. *Physics*. John Wiley and Sons, New York, 1966.

57. Samuel P. Harbison. *Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1992.

58. Samuel P. Harbison and Guy L. Steele, Jr. *C: A Reference Manual*. Prentice Hall, Englewood Cliffs, NJ, third edition, 1991.

59. Cay S. Horstmann. *Mastering C++: An Introduction to C++ and Object-oriented Programming for C and Pascal Programmers*. John Wiley and Sons, New York, 1991.

60. Richard D. Jenks. A primer: 11 keys to new Scratchpad. In John Fitch, editor, *EUROSAM 84: International Symposium on Symbolic and Algebraic Computation*, volume 174 of *Lecture Notes in Computer Science*, pages 121–147, Springer-Verlag, Berlin, 1984.

61. Richard D. Jenks and Robert S. Sutor. *Axiom: The Scientific Computation System*. Springer-Verlag, New York, 1992.

62. Max E. Jerrell. Function minimization and automatic differentiation. In Norman Meyrowitz, editor, *Proceedings Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'89)*, pages 160–173. Association for Computing Machinery, October 1989.

63. Jay Kappraff. *Connections: The Geometric Bridge between Art and Science*. McGraw-Hill, New York, 1991.

64. Brian M. Kennedy. The features of the object-oriented abstract type hierarchy (OATH). In *USENIX C++ Conference Proceedings*, pages 41–50. USENIX Association, April 1991.

65. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1978.

66. Donald E. Knuth. Structured programming with go to statements. *ACM Computing Surveys*, 6(4):261–301, December 1974. Reprinted in [122].

67. Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, second edition, 1981.

68. Andrew Koenig and Bjarne Stroustrup. C++: As close to C as possible—but no closer. *The C++ Report*, 1(7), July 1989.

69. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math Software*, 5(3):308–323, September 1979.

70. Charles L. Lawson and Richard J. Hanson. *Solving Least Squares Problems*. Prentice Hall, Englewood Cliffs, NJ, 1974.

71. L. R. Lines and S. Treitel. A review of least squares inversion and its application to geophysical problems. *Geophysical Prospecting*, 32:159–186, 1984.

72. Mark A. Linton. Encapsulating a C++ library. In *USENIX C++ Conference Proceedings*, pages 57–66. USENIX Association, August 1992.

73. Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

74. Stanley B. Lippman. *C++ Primer*. Addison-Wesley, Reading, MA, second edition, 1991.

75. Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. In Andreas Paepcke, editor, *Proceedings Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93)*, pages 16–28. Association for Computing Machinery, October 1993.

76. Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society of Industrial and Applied Mathematics*, 11(2):431–441, June 1963.

77. Bruce Martin. The separation of interface and implementation in C++. In *USENIX C++ Conference Proceedings*, pages 51–63. USENIX Association, April 1991.

78. Michael Metcalf. FORTRAN 90—a summary. *International Journal of Modern Physics C*, 1(2):193–206, September 1990.

79. Michael Metcalf and John Reid. *Fortran 90 Explained*. Oxford University Press, New York, 1990.

80. Bertrand Meyer. Reusability: The case for object-oriented design. *IEEE Software*, 4(2):50–63, March 1987.

81. Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1988.

82. Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, Reading, MA, 1992.

83. Scott Meyers and Moises Lejter. Automatic detection of C++ programming errors: Initial thoughts on a lint++. In *USENIX C++ Conference Proceedings*, pages 29–40. USENIX Association, April 1991.

84. R. B. Murray. Building well-behaved type relationships in C++. In *USENIX C++ Conference Proceedings*, pages 19–30. USENIX Association, 1988.

85. Lee R. Nackman and John J. Barton. Base-class composition with multiple derivation and virtual bases. In *USENIX C++ Conference Proceedings*. USENIX Association, April 1994.

86. Oscar Nierstrasz. Regular types for active objects. In Andreas Paepcke, editor, *Proceedings Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93)*, pages 1–15. Association for Computing Machinery, October 1993.

87. David L. Parnas. Information distribution aspects of design methodology. In *Proceedings of the 1971 IFIP Congress*, pages 339–344, Amsterdam, North-Holland, 1972.

88. David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

89. David L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, March 1979.

90. David L. Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, March 1985.

91. P. J. Plauger. *The Standard C Library*. Prentice Hall, Englewood Cliffs, NJ, 1992.

92. Terrence W. Pratt. *Programming Languages: Design and Implementation*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1984.

93. William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, second edition, 1992.

94. L. B. Rall. The arithmetic of differentiation. *Mathematics Magazine*, 59:275, 1986.

95. Louis B. Rall. *Automatic Differentiation*. Springer-Verlag, Berlin, New York, 1981.

96. Paul Roman. *Some Modern Mathematics for Physicists and Other Outsiders*, volume 1. Pergamon Press, New York, 1975.

97. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

98. Dan Saks. The return type of virtual functions and the C standard library. *The C++ Report*, 4(7):61–63, September 1992.

99. B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines–EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, second edition, 1976.

100. Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In Norman Meyrowitz, editor, *Proceedings Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'86)*, pages 38–45. Association for Computing Machinery, November 1986.

101. Alan Snyder. Inheritance and the development of encapsulated software components. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. The MIT Press, Cambridge, MA, 1987.

102. Ian Sommerville. *Software Engineering*. Addison-Wesley, Wokingham, England, third edition, 1989.

103. V. Srinivasan, L. R. Nackman, J.-M. Tang, and S.N. Meshkat. Automatic mesh generation using the symmetric axis transformation of polygonal domains. *Proc. IEEE*, 80(9):1485–1501, September 1992.

104. David Straker. *C Style: Standards and Guidelines: Defining Programming Standards for Professional C Programmers*. Prentice Hall, Englewood Cliffs, NJ, 1992.

105. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

106. Bjarne Stroustrup. What is object-oriented programming? *IEEE Software*, 5(3):10–20, May 1988.

107. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, second edition, 1991.

108. Bjarne Stroustrup. A history of C++: 1979–1991. *ACM SIGPLAN Notices*, 28(3):271–297, March 1993.

109. Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.

110. Russell H. Taylor. Planning and execution of straight line manipulator trajectories. In Michael Brady, John M. Hollerbach, Timothy L. Johnson, Tomás Lozano-Pérez, and Matthew T. Mason, editors, *Robot Motion: Planning and Control*, pages 265–286. The MIT Press, Cambridge, MA, 1982.

111. Steve Teale. *C++ IOStreams Handbook*. Addison-Wesley, Reading, MA, 1993.

112. D'Arcy Wentworth Thompson. *On Growth and Form*. Cambridge University Press, Cambridge, abridged edition, 1961.

113. David Ungar. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167. Association for Computing Machinery, April 1984.

114. Peter Wegner. Dimensions of object-based language design. *ACM SIGPLAN Notices*, 22(12):168–182, December 1987. Special Issue: Proceedings Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'87).

115. Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP '88*, pages 55–77, 1988.

116. Rebecca Wirfs-Brock and Brian Wilkerson. Object-oriented design: A responsibility-driven approach. In Norman Meyrowitz, editor, *Proceedings Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'89)*, pages 71–75. Association for Computing Machinery, October 1989.

117. Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, Englewood Cliffs, NJ, 1990.

118. Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 16(2):221–227, April 1971. Reprinted in [121].

119. Niklaus Wirth. On the composition of well-structured programs. *ACM Computing Surveys*, 6(4):247–259, December 1974. Reprinted in [122].

120. Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, Redwood City, CA, second edition, 1991.

121. Edward Yourdon, editor. *Writings of the Revolution: Selected Readings on Software Engineering*. Yourdon Press, New York, 1982.

122. Edward N. Yourdon, editor. *Classics in Software Engineering*. Yourdon Press, New York, 1979.

123. Marvin V. Zelkowitz, editor. *Selected Reprints in Software*. IEEE Computer Society Press, Washington, DC, third edition, 1987.

124. O. C. Zienkiewicz. *The Finite Element Method*. McGraw-Hill, London, third edition, 1977.

INDEX

Page numbers in italics refer to the definition of the concept or class. Entries labeled "example of" refer to code defining a class or introducing a language construct; entries labeled "example of use" refer to code using the class or language construct in an example.

Special Symbols

!, 24, 31
!=, 24, 31, 45, 504
#, 15
%, 23, 31, 205
%=, 31
&, 27, 31, 43, 44, 49, 51, 79, 125, 127, 160, 446, 512
&&, 24, 31
  example of use, 158
&=, 31
(), 31
*, 23, 31, 43, 49, 62, 77, 125, 160, 427, 431, 480
**, 23
*/, 14
*=, 31
  example of use, 316
+, 23, 31, 49, 160, 480
++, 23, 31, 41, 49, 165, 480, 491
++*
  example of use, 434
+=, 31
  example of use, 129
,, 42
-, 23, 31, 49, 160, 480
--, 23, 31, 49, 165, 480, 491
-=, 31, 49
->, 31, 49, 89, 430; *See also* member—selection operator.
->*, 31, 71, 514; *See also* pointer—to member operator.
., 31, 88, 160
.*, 31, 71, 160, 513, 514; *See also* pointer—to member operator.
/, 23, 31, 480
/*, 14
//, 13, 68
/=, 31
:, 147, 236, 268
::, 31, 71, 91, 110, 112, 160, 166, 267, 286, 293, 513, 549
;, 86
<, 24, 31, 45, 136
<<, 15, 27, 28, 29, 31, 68, 72, 74, 171, 258, 552
  concatenating, 68
<<=, 31
<=, 24, 31, 45
<>, 101, 104
=, 23, 26, 31
  for assignment, 26
  for default argument, 130
  for initializer, 38
  used for initialization, 149
==, 23, 24, 31, 45, 504
  example of use, 93
>, 24, 31, 45, 136
>=, 24, 31, 45
>>, 15, 27, 28, 31, 68, 72, 171, 172, 174, 258, 552
  concatenating, 68
>>=, 31
?:, 160; *See also* conditional operator.
[], 31, 78, 367; *See also* operator[](), projection—operator.
  example of use, 377
  in new and delete, 78
\", 25
\', 25
\?, 25
\\, 25
^, 27, 31
^=, 31
{}. *See* braces.
|, 27, 31
|=, 31
||, 24, 31
  example of use, 273, 310
~, 27, 31, 96; *See also* destructor.
□. *See* composition law.

0

0, 24, 76
  as null pointer, 44
=0. *See* pure virtual function.
0x, 26

A

\a, 25
_AXPY, 136
A, *167*, *357*, *447*
  as public base, 357, *605*
Abelian, *475*
AbelianGroupCategory<T>, *490*, 505
  as public base, 491
AbelianMonoidCategory<T>, 488
  as public base, 490
AbelianSemiGroupCategory<T>, *486*, 487, 501, 502
  as public base, 500
Abelson, H., 535
abort(), 182
abs()
  example of use, *36*, 93, 133
abstract algebra, 505
  programming device for, 473
abstract base class, *243*, 262, *329*
  using to avoid errors, 276
abstract class, *262*
abstract factory, *262*; *See also* object—factory.
abstraction, 4, 120, *206*, 207, 218, 260
  array shape, 315
  arrayness, 313
  class for, 207
  collections, 303
  in communication, 3
  of control flow, 535
  creation versus use, 242
  current-voltage curve, 250
  design cost of, 218
  distinct from encapsulation, 212, 218
  essential aspects of, 212
  expressing, 3
  failure of, 212
  finding, 207, 218
  ignoring template expansion, 350
  importance of, 206
  information loss, 340
  interface base class, 234
  in a nonmember function, 233
  and polymorphism, 327
  pure virtual functions and, 242
  of relationships, 212, 218
  represented by interface base class, 236
  secrets of, 207
  with subroutines, failure of, 207
  supported by iterators, 413
  violation of, 212
  voltmeter, exercise, 263
Acceleration, 495, 499
accept function, *344*
access, 285
  friends, 167
  to member data, 89
  to members, 87
  members of other objects, 92
  nested class, 110
  preventing with encapsulation, 207
  specifier, 290
access control
  adjustment, 293
  after dominance and overloading, 290
  base class members, 290
  and derivation (table), 290
  for derived classes, 292
  encapsulation, 290
access declaration, *293*, 299
  example of use, 308, 374, 380
  limitations on, 294
  and overloaded names, 294
AccessedArray1d<T>, 406
AccessedArray2d<T>, 403, *404*
AccessedConstArray2d<T>, 405
AccessedPhysicalDataBrowser<double>, 586
AccessedPhysicalDataBrowser<T>, 614
accessor, *398*, 403, *415*
  compared to reference, 404
  example of use, 585
Acme130, 225, *227*, 228–231, 233–236, 246, 247, 251, 263, 271, 272, 274, 281, 291, 293, 299, 334, 340, 342
Acme130Simulation, *251*, 253, 257, 263
Acme130_Acs, *293*, 294, 295, *296*
Acme130_Fwd, *278*, 279, 281
Acme130_GC, 341, *344*
Acme130_GCTC, *342*, 343
Acme130_GIData, *281*–289, 292, 294, 301
  as private base, 285
Acme130_NoV, *294*
  as public base, 294
Acme130_VS, *235*, 236–244, 334, 335, 340
Acme130_VS_GI, *244*–246, 278, 301
Acme130_VS_GI_GC, *247*, 250, 256, 263, 271–281, 298
  as public base, 271
Acme140, *271*, 272–276, 299, 301
Acme140_Fwd, 301
Acme230, *285*, 286–289, 294–296, 301
Acme230_NoV, *294*, 295
actual argument, *57*, *80*, 125
acyclic graph, 239
Ada
  and generic types, 327

compared to C++, 6
adaptability
via system of classes, 363
Adcock, J. L., 11
Add, *501*, 502
AddFunctional<Domain>, *522*, 523
additive, *475*
AdditiveArray, *500*, 501
address, 43, 46
address-of operator, *43*, 44
advance(), 408
aggregate, *420*
aggregated object, *421*
aggregation, *420*; *See also* containing aggregation, controlling aggregation, referential aggregation, and shared—aggregation.
controlling, example of, 461
kinds of, 421
referential, example of, 461
Aki, K., 575, 603, 604, 613
alert (bell), 25
algebra, *476*
algebra with unit, *476*
example of, 564
algebraic category
example of use, 591
selection of, example, 496
single set, additive, table, 483
single set, multiplicative, table, 481
algebraic structure, 505
decision tree, 476
matrix example, 564
programming device, 478
single set properties, table, 474
two set examples, table, 476
algebraic structure categories, *473*
AlgebraWithUnitCategory<V, S>, *492*
as public base, 564
algorithms, 613
allocation, *175*
allocator, 38
ambiguity
avoiding, 153, 155, 482
with conversions, 153
from duplicate base classes, 289
duplicated base class member function, 283
of representation, information hiding, 230
solved by hiding, 288
state-representation, 230
ambiguous
function call, 133, 135, 137
and, bitwise, 27
.AND., 24
Anderson, E., 370, 451, 452, 467, 559, 579
angle brackets. *See* template.
ANSI C, 67
called from C++, example of, 540
documentation, 578
string class, 578
string functions, 542
ANSI standard
function template, 139
name spaces, 173
APL, 535
compared to C++, 7
Apple, *359*
application
of functions to collections, 533
argc, 532
argument, 52
actual, 125
altering, 124
array as, 129
and const member function, 145
default, 130
formal, 125
initialization of, 58, 81, 125
initialized with copy constructor, 126
LAPACK commonality, 455
matching, function call operator, 514
as mathematical domain, 122
name omitted to suppress warning, example of, 394
non-const reference, example of, 595
not allowed for conversion operator, 152
passed by reference, 59
passed by value, 58
reference, 127
relation of formal and actual, 125
swapping, example, 127
template, 136
type conversion of, 126
argument declaration, *122*
argument matching, *132*
best match, 132
const and, 134
conversion operator, 152
exact, 133
example of, 91
function template and ANSI, 139
function templates, 137
and member function, 143
multiple arguments, 135
operator, 161
with promotions, 134
and single argument constructor, 151

argument matching *(cont.)*
standard conversion, 134
template wins, 325
throw-catch, 110
with trivial conversions, 133
user-defined conversion, 135
argument type
void, 131
arguments
swapping, 81
argv, 532
arithmetic
on pointers, 45
arithmetic operator
table, 23
arity, *160*
ARM, 5, 11
marginal notes, 8
Arnold, R. S., 619
array, *40*, *155*, *312*, *318*
1-offset origin, 117
as actual argument, 59
allocating with new, 78
allocation, example of, 99
allowed conversions to reference, 385
with arithmetic distributed to elements, 500
assignment, 96, 99
example of use, 366
to scalar, 99
begins with element 0, 62
bounds checking, example of, 109
built-in, 9, 62, 312
as parametric type, 330
and pointer, 420
character, initialization of, 50
class, 95, 97
of class objects, 86
class, no perfect, 312
class template, example of, 101
classes, common data in, 266
column, 366
compatible with FORTRAN, 369
concrete, *313*, 363
design criteria, 368
examples of use, 364
contiguous element storage, 370
conversion to reference, example of, 385
copy, 96
copy constructor, 98
of counted-object pointers, example of, 438
deallocating with delete, 78
declaration (table), 42
derivation from concrete, example of, 457
destructor, example of, 98
dimensionality function, implementation, 32
dimensionality, defined, 312
dynamically allocated, 189
element, *40*
access, 95, 366
access, through pointer, 45
in character string, 50
modifying, 134
pointer to, 45
element offset, 371
example of, 374
element storage, 371
element storage choices, 368
element type via typedef, 321
exception classes, 316
flexibility, defined, 312
formed (runtime sized), 313
function applied to elements, 533
as function argument, 129
initialization, 41, 42
dynamic, 189
of interface pointers
example of, 444
function application and, 535
interface
dimension in name, 319
dimension independent, 318
element type independent, 318
returned by reference example, 589
interfacer template, 401
interfacer template, example of use, 402
iterator, 406
list of, built at runtime, 397
maximum element, example, 364
maximum possible subscript, 315
member datum, example of, 97
member function, example of, 97
member functions, table, 368
memory management, 96, 97
multidimensional, 41
packed storage, 559
and pointer, 44, 95
of pointers to interface base class, 250
projection, *364*, 366
return type, 367
reference
as basis for projection, 379
const-like, 380
created from array, 385
dangling, 387
design criteria, 380
for matrix transpose, 568

from array conversion, 385
non-const-like, 382
object sketch, 386
resizing, example of, 99, 102
rigid (compile-time sized), 312, 322
row, 366
row major, contiguous formed, see FormedArray, 314
shape, *312*, 371
exception, example of, 383
implementation of, 372
size, 96
size overhead, 372
storage allocation, 371
storage layout, 313
stride, 370, 389
subscripting, 98
implementation of, 377
with operator()(), 318
system of classes, 363
advantages of, 413
introduction, 311
terminology, 312
variable size, 47
zero dimensional (element), 367
Array1d<float>, 335, 336, 337, 338
Array1d<T>, *319*, 323, 324, *327*, 328, 336, 338, 589, 596, 598, 600
as public virtual base, 405, 588, 599
Array2d<double>, 320, 322, 350
Array2d<int>, 322, 350
Array2d<T>, *319–321*, 323, *324*, 349, 350, 398, *400–406*, 413, 414, 461
as public virtual base, 322, 401, 404, 461, 464
Array<2, float>, 318
Array<T>, 352
ArrayBrowser2d<T>, *412*, 417
arrayCopy(), 416, 601
example of use, 600
ArrayErr, *316*
ArrayIterator2d<T>, 417
ArrayShape, 314, *315*, 316, 318, 321, 323, 324, 328, 338, 398
as public virtual base, 318, 319, 321, 323, 324, 398
ArraySizeError, *108*, 109, 110, *258*
ArrayStepper2d, *412*, 413
as public base, 412
arrow operator. *See* ->, member—selection operator.
assignment, 69, *148*
array, 99
array arguments and, 130
array, example of use, 366
avoiding self-overwrite, 99, 429
cascaded, problems with, 173
class objects, default, 91
enumeration, 71
equals used for initialization, 149
to formal arguments, 58
function call on left-hand side, 79, 144
function pointer, overloading and, 512
to reference, 51, 79
to reference returned by a function, 132
static member datum, 94
versus equality test, 23
assignment operator, *26*, 31
example of, 99, 383, 386, 395, 549, 591
generated, 163, 191
interface base class, example of, 401
not inherited, 268
other types of arguments, 163
overloaded, example of, 545
private, to prevent use, 400
programmer-defined, 191
selecting template expansion, example of, 602
association
between variable and object, 20
function argument and object, 81
name and object, 18, 69, 86
variable and dynamic object, 188
associative, *474*
associativity, 30, *71*, 474
programmer-defined operator, 100, 160
Atkins, M. C., 195
Atom, *442*, 443, 447
as public virtual base, 442
Atom*, 442
attribute, 38
Atwood's machine, 494
AtwoodsMachine, *494*, 495, 499
augmented matrix
class for, 605
automatic lifetime, *176*
automatic differentiation, *583*, *589*
example of use, 594, 600, 610
for second derivatives, 613
automatic object, *185*
lifetime, 176
pointer to, 192
for resource management, 187, 195
automatic type conversion, 54
AveragingFloatArray, 301
Axiom, 261
axpy function template
example, 136, 138

B

\b, 25
B, *167*, *262*, *357*, *447*
  as public base, 262
backslash, 25
backspace, 25
Backus, J., 139
Bai, Z., 370, 451, 452, 467, 559, 579
Barton, J. J., 300, 312
base class, 236, 298
  abstract, 329
  access control, 290
  choosing virtual, 301
  for collection of objects, 343
  constructor call, example of, 271
  diagrams, 239
  equality testing, 352
  example of, 235, 236, 244, 247, 251, 252, 254, 255, 258, 262, 266, 271, 278, 281, 285, 293, 294, 307–309, 316, 324, 342, 345, 353, 357, 359, 373, 374, 378, 380, 382, 390, 391, 402, 412, 430, 436, 439, 447, 457, 479, 484, 488–491, 493–495, 497, 498, 500, 503, 517, 518, 522, 529, 541, 555, 560, 563, 564, 566, 568, 585, 588, 591, 593, 599, 604, 605
  implementation versus interface, 266, 271
  implementation, example of, 277
  initialization, example of, 284
  interface. *See* interface base class.
  interface versus implementation, 266, 271
  list of pointers to, 340
  member function as constraint, 275
  multiple duplicated, 283
  multiple shared (virtual), 282
  must make virtual decision, 275
  parameterized by class template, example of, 484
  parameterized by derived class, 355, 480
    example of, 353
  public, 236, 266
  reference
    example of use, 270
    initialized from conversion, 338
  represents, 234
  specifying, 236, 244
  for template class, 307
  template derived from template, 311
  template, example of, 352
  template, for derived class template, 309
  templatized implementation, 323
  templatized interface, 313
  templatized, of template class, 310
  uses, 236
  without member data, 236
base class composition, *294*, 299
  diamond-shaped class DAG, 296
  example of, 588, 599
  precluded by downcast, 343
  templatized, 323
base classes, relationships among. *See* class DAG.
base initializer, *268*
base subobject, *266*
  private, 282
base type, *335*, *336*, 358
Basic Linear Algebra Subroutines, 136; *See also* BLAS.
Bathe, K.-J., 218
behavior, 4, *229*
  related to state, 229
Bergland, G. D., 10
best match, 132
binary data, 62
BinaryFunctional<Domain>, *522*, 523
  as private base, 522
binding
  object to variable, 69
Bischof, C., 370, 451, 452, 467, 559, 579, 589
bit pattern, 26
bitwise operator, 31
  table, 26
Blaha, M., 10, 261
BLAS, 559, 563
  level 1, 136
Blas1Subroutines, *166*, 167, 564
Blas2Subroutines, 564
  as public base, 563
Blas3Subroutines, *563*, 564
BlasErr, *555*, 579
block, *32*
block scope, *194*
blueprint for objects
  class as, 227
Booch, G., 6, 10, 218, 261, 262, 358
Boolean, *151*–153, 172, 173, 470
  example of use, 307, 461
bottom-up, *10*
Boyle, J. M., 451
braces, 13, 32
break, 37
BritishConstants<T>, 495
  as private base, 495
Brooks, Jr., F. P., 10, 207
Brown, J. A., 7, 535
browser, *409*, *547*; *See also* iterator.

concrete array, example of, 410
equivalent to subscripting, 412
example of, 586
interface, example of, 584
interfaced arrays, 412
BrowserType, 409
buffer
flushing I/O, 68, 83
built-in
array, 9, 98
and pointers, 420
argument, const, 130
to pointer conversion, 133
allocation, 99
avoiding, 190
delete and, 98
dynamic, 97
problems with, 312
used in array class, 322
passed by reference, 129
assignment operator, 164, 173
character array, example of use, 542
class object used like, 86
conversions, 338
dynamic object, 188
initialization of, 189
exponentiation operator, none, 104
member datum
copies of, 148
initialization of, 147
uninitialized, 148
number, 21
object
as argument, 126
conversion from class object, 152
conversion to class object, 152
and functions with internal linkage, 124
initialization of, 146, 176, 184
I/O, 171
as member datum, 94
pass by value, 129
versus class object, 86
operator, 27, 159
related to programmer-defined, 161
versus programmer-defined, 100
parametric type, 330
const, 332
pointer
copied by value, 164
and dynamic array, 150
as member datum, 149, 192, 434
and memory management, 100, 150
problems with, 419
referential aggregation, 423
to function, 511
returned from inline function, 56
type, 70, 330
as template argument, 136
mimicked by classes, 100
types, same as C, 541
built-in parametric type, *330*, 356
BuiltInPtr<T>, 446
Bunch, J. R., 136, 451

C

C, *167*
C analog
>>, 68
array storage format, 313
built-in type, 70
character constants, 70
declaration, 75
#define, 75
enumeration, 70
free(), 77
fscanf(), 68, 72
function name, 122
function with no arguments, 80
global variables, 207
functions, 80
macro preprocessor, 303
macros, 75
malloc(), 77
NULL, 76
operators, 71
pointers, and const, 76
pointers, for pass by reference, 80
sprintf(), 75
sscanf(), 75
stdio.h, 68
struct, 115
type of 'x', 70
C language
C++as superset of, 7
call to C++ function, 541
compared to C++, 6, 67, 540
functions called from C++, 540
"C", 540, 553
C++ standardization, 5
C1, *262*
as public base, 262
C2, *262*
cache behavior, 55, 82
caching, *180*
exercise, 536

CachingFunction<Domain, Range>, 536
Calder, P. R., 262
call
  by const reference, *128*
  by reference, *59*, *81*, *127*, 128
  by value, *126*
    to avoid side-effects, 125
Cardelli, L., 327
Cargill, T., 11, 300, 578
carriage return, 25
case, significance of, 19
cast, *342*
  heap pointers, 77
catch, 107–110, 140, 185, 257, 259, 360
  example of, 259
category, 260, *261*
  function-structure, 354
  function-structure, example of, 480
  implementation, 298
  interface, 231, 234
  Smalltalk, 261
  template, 314
category-provides functions, 482
CAXPY, 136
cerr, 27, 72
  example of use, 32
chain rule, 590
change
  planning for with typedef, 40
changing representation, 229
char, 21–24, 70, 134, 554
CHARACTER, 554
character
  array of, 25
    initialization of, 50
  comparison, 25
  constant, 24
    size of, 70
  converted to integer, 24
  string, 15, 25, *50*, 68, 542
    command line arguments, 60
    example of use, 177
    terminated by null character, 50
  unprintable, 24
character set
  effect on character comparison, 25
CHARACTER*1, 22
CHARACTER*n, 554
char[n], 554
checked pointer, *448*
CheckedFloatArray, *266*, 267–270, 276, 301, 310
CheckedSimpleArray<double>, 327
CheckedSimpleArray<float>, 310, 349
CheckedSimpleArray<int>, 310, 349
CheckedSimpleArray<T>, 107–109, 258, 266, *309*, 310, 327, 349, 350
chi-squared (χ^2), *587*
Cholesky factorization, 456
Chow, T. S., 10
cin, 15, 27, 28, 68, 72
  logical tests of, 35
Circle, *145*, 146, 149
circular doubly linked list, *174*
circular linked list, 174
class, *4*, *85*, 86, 101, 104, 115, 136, 168, 227
  abstract, 334
  as abstraction, 207
  as blueprint for objects, 227
  base, for implementation reuse, 266
  body, *86*
    of template, 101
  category, *261*
  concrete, 270, 363
  const-like reference or interface, 415
  coupling, by paired friend and private constructor, 410
DAG. *See* class DAG.
declaration, *85*, 115
definition, *85*, 115, 227, 330
function. *See* static—member function.
hierarchy, *4*, *261*; *See also* class DAG.
instance of, 86
instances, as set of objects, 473
in an interface category, 235
  lattice, *261*; *See also* class DAG.
matrix, related to FORTRAN subroutines, 457
member access operator, *88*, 89, 165
multiple meanings of, 241
name, 91, 96, 110
  uniqueness of, 330
nested, 110
with no data, example of, 226
with no objects, 240
object, *86*, 93
  pointer type, 419
  with restricted values, 483
  scope, *178*, 286
  similarity to built-in types, 100
  as template argument, 101
  as type, 89
  use versus design, 230
  versus typedef, 208
class DAG, *239*, *261*
  attempted extension without virtual bases, 295
  base class composition, 295
  concrete array projection, 388

ConcreteFortranArray2d<Subscriptor, T>, 371
for derivation of template class from common base, 308
diamond-shaped, 296 .
dominance, example of, 288
duplicate base classes, 283, 289
interfaced array, 314, 398
iterated nonlinear equations, 595
leaves, example of, 314
for multiple derivation, 245
not reflected in smart pointer, 447
public extension of interface, 295
shallow versus deep, 297
shorthand for templates, 311
showing class is-usable-as relationship, 271
single-set algebraic categories, 481
for template class derived from template class, 310
templatized base class composition, 323
two-set algebraic structures, 492
virtual base class, 282
and multiple derivation, 286
well-formed, 289, 294, 298
class template, *101*, *304*
declaration of, 101
definition of, 101
empty body, example of, 347
instance of, 101
not a type, 346
for simulating a function template, 488
with two parameters, example of, 484, 591
versus base class, 326
versus template class, 101, 304
cleanup object, 176
Clements, P. C., 207
Clenshaw's recurrence formula, *537*
client, *143*
of an abstraction, 207, 212
array interface, 319
of base class, 270
class, 250
choosing schizophrenic behavior, 270
of a class library, 194, 208
function template, 305
function, *246*, 250
access to base class members, 282
taking object not reference, 275
nonmember, 325
of interface base class, 246
of template versus client of interface, 326
using pointer to interface, 420
view of inheritance, 275
virtual function, 260
clone function, 442
example of, 516, 521
example of use, 444
problem with, 443
return type, 447
clone(), 443, 444, 447, 516, 517, 536
CloneableObjPtr< IsoFunctional<Domain> >, 536
CloneableObjPtr<Atom>, 447
CloneableObjPtr<Base>, 445
CloneableObjPtr<Derived>, 445
CloneableObjPtr<T>, 425, *443–449*, 516, 520
code duplication, 55, 82
to avoid coupling, 547
code reuse, 298
projections of arrays and array references, 393
code size, 55, 82
Cohen, J., 195, 446
Coleman, T. F., 7
collection
of base class pointers, 340, 341
beyond arrays, 415
function applied over, 533
iteration, 406
of pointers to interface base class, 250
Colloms, M., 226
colon, in base class specifier, 236, 244
column(), 367
column-major, array storage layout, 42, 313, 364, *370*
FORTRAN compatible, in C++, 553
comma operator, 31, 42
Command, 196
command line arguments, 60, *532*
comment, 13, 68
content of, 16
common functions, as potential virtual functions, 233
common implementation, 100, 298
example of, 277
inherited, example of, 377
commonality, 260
are needed together, 340
data, example of, 266, 277
function specification, 231, 315, 326, 327
function-structure, 354
example of, 478
has-structure-of (templates), 303
implementation, 306, 326
interface base class, 236
matrix and its factors, 464
member data and function implementation, 298
name, 305, 348

commonality *(cont.)*
name
example of, 364, 465, 557
example of use, 401, 409, 487
typedef for, 320
used by nonmember function, 325
user-must-define functions, 483
nonlinear equations, 594
related subroutine arguments, 455
shared operator, example of, 516
structural, 327
structure and members combined, 307
structure of implementation, 306
trade-offs in expressing, example of, 396
and types, 329
wrapping concrete arrays in interfaces, 397
commutative, *475*
commutative ring, *475*
with unit, *475*
commutativity, 474
ComparableFormedArray1d<int>, 354
ComparableFormedArray1d<T>, *353*, 354
ComparableRigidArray1d<T,n>, 353
comparison operator, example of
compilation time
effect of inline function on, 55, 82
compile error, 329
array size mismatch, 366
private member of base class, 292
from type checking, 233, 331
from units mismatch, 495, 500
complement, 27
complex, 21, 138, 150, 153, 160, *161*, *162*, 339, 478, 554
complex numbers, 21, 478
example, 153, 166
COMPLEX*16, 554
complex.h, 478
ComplexDouble, *153*, *154*, 155, *156*
ComplexFloat, 135, *153*, *154*, *156*, *166*, *339*, 478, *479*, 480, 485, 486, 506, 513
ComplexInt, 153, *154*, *156*
ComplexRoots, *111*, 112
composition law, 474
associative, 474
binary, 474
commutative, 475
external, 476
computer algebra, 261
concrete array, *363*; *See also* array—concrete.
concrete class, 270, *275*
array, defined, 313
candidate for, 414
derivation, example of, 373
template, inheritance and, 309
ConcreteArray1d<Atom*>, 443
ConcreteArray1dRef
as public base, 391
ConcreteArray1dRef <ConcreteColumnMajorProjectionSubscriptor<1>,T>, 392
ConcreteArray1dRef<Subscriptor, T>, 388, 391
ConcreteArray2d<Subscriptor, T>, 371, 374, 381, 385–388, 391, 393, 402, 415, 560, 580
ConcreteArray2d<Subscriptor,T>, 393
ConcreteArray2d<T>, *374*, 377, 402, 423
as public base, 378, 560
ConcreteArray2dConstRef<Subscriptor, T>, 376, *380*, 385, 399
ConcreteArray2dRef<Subscriptor, T>, 376, *382*, 393
as public base, 568
ConcreteArrayBrowser<ConcreteArray>, *410*
ConcreteArrayIterator<ConcreteArray>, *411*
ConcreteArrayProjection1d<ConcreteColumnMajorSubscriptor<2 >, T>, 392
ConcreteArrayProjection1d<Subscriptor, T>, 388, *391*, 393–396, 406
ConcreteArrayShape<2>, 371
ConcreteArrayShape<ndim>, *372*, 388, 390
as private base, 390, 391
as public base, 373
ConcreteBlas1d<T>, 565, 567, 580
ConcreteBlas2d<double>, 607
ConcreteBlas2d<T>, *564*–568, 580
ConcreteBlasProjection1d<T>, 566
ConcreteColumnMajorProjection1d<T>, 398
ConcreteColumnMajorProjectionSubscriptor<1>, *391*
ConcreteColumnMajorProjectionSubscriptor<ndim>, *390*
ConcreteColumnMajorSubscriptor<1>, 374
ConcreteColumnMajorSubscriptor<2>, 370, 371, 375, 393
ConcreteColumnMajorSubscriptor<ndim>, *373*, 389, 393
ConcreteElasticArray2d<T>, 415
ConcreteFormedArray1d<T>
as public base, 500
ConcreteFormedArray2d<T>, 398, 402
ConcreteFortranArray1d<T>, 580
ConcreteFortranArray2d<double>, 371, 393
ConcreteFortranArray2d<T>, 371–*378*, 388, 391, 392, 398, 402, 457, 458, 461, 555, 559, 565–568, 580
as public base, 457, 564, 566
ConcreteFortranSymmetricPackedArray2d<T>, 457, *560*
as public base, 457

ConcreteRigidArray1d<T>
as public base, 503
ConcreteRigidArray2d<T, n0, n1>, 415
ConcreteRowMajorSubscriptor<2>, 402, 406
ConcreteRowMajorSubscriptor<3>, 370
ConcreteStrides<ndim>, *389*, 390
as private base, 390, 391
ConcreteVector3d, 506
condition number, *571*, 613
conditional operator, 31, *33*
example of use, 271, 308, 322, 427
unreadable, 33
const, 39, 46, 51, 76, 79, 125, 128–130, 134, 143–145, 148, 150, 164, 166, 173, 190, 323, 332, 340, 359, 369, 447, 448, 554
member function, 144
object, example of use, 323
object, reference to, 51, 79
and pointer, 46
pointer, cannot be deleted, 189
reference, 51, 79
const member datum
initialization, 148
const member function, 173
const reference, 128
const reference argument
and const member function, 145
const type, 331
const variable
initialization of, 39, 76
constant, 38; *See also* const.
machine dependent, 574
physical, 75, 179
symbolic, with class scope, 173
constant expression
as template argument, 101
constant object, *39*, *75*
ConstantFunctional<double>, *523*, 531
ConstArray2d<T>, *398*, 399, 402, 405, 413
as public virtual base, 400
ConstConcreteArrayProjection1d<Subscriptor, T>, 388
ConstConcreteBlasProjection1d<T>, *566*
ConstProjectionT, 367, 368
constraint
base class member function as, 275
as interface base class functions, 242
construction, *176*
of static object, 181
construction operator, *339*
construction, order of
automatic objects, 185
of static local objects, 182
of static nonlocal objects, 182, 183
of static object, example of, 181
constructor, *88*, 96, 113, *145*, 176
base class
called by derived class, 268
example of, 278
base initialization, example of, 268
built-in type, 146
call
example of, 91, 268
explicit, 146, 379
example of, 592
implicit, 88, 146
preventing, 376
calling pure virtual function, 261, 356
class template derived from class template, 310
copy
and copied-object pointer, 425
and counted-object pointer, 425
example of, 379
generated, 148, 276
default, *148*, 268
example of, 520
generated, 269
needed for arrays, 189
static objects, 194
empty body, example of, 269
example of, 91, 102, 214, 228, 461, 549, 600, 607, 608
function name, 91
function template, example of, 310
generated default, 148, 269
generated, and copied-object pointer, 425
generated, appropriate, 431, 435
generated, preventing, 376
as initializer, 145
initializing base subobject, 268
initializing reference, example of, 248
not inherited, 268
order of call, base and members, 268
overloaded, 91, 145
example of, 91, 519, 545, 591
private, 269
example of, 376, 410, 462, 546, 550
for friend class, 410
protected
example of, 391
to prevent object creation, 292
required for class with implementation base, 271
single-argument, 146, 148, 150, 151, 156, 172, 339

constructor *(cont.)*
single-argument
and argument matching, 127
as conversion, 151
example of, 152, 568
static members as alternative to, 166
taking class arguments illegal, 173
virtual, 442; *See also* clone function.
without arguments, 146
container class
template for, 305
containing aggregation, *420*
controlling object lifetime, 421
sharing object lifetime, 422
continue, 37
contract between class and client, 208
control structures, *30*
controlling aggregation, *421*
copied-object pointer for, 427
and interfaces, 441
convenience of use
via system of classes, 363
conversion, 150
causing ambiguity, 153
const reference argument, 129
const type, 332
derived-class reference to base-class reference, 338
explicit, 158
of function name to pointer, 511
global operator, 161
implicit, 158
indirect view versus value change, 333
int to enum not allowed, 26
and object initialization, 151
preventing unwanted, example of, 593
runtime error, 155
standard, 134
trivial, 133
user-defined and argument matching, 135
using to add checking, 157
using to add state, 157
via single argument constructors, 172
conversion function, *152*
example of, 308
explicit versus implicit, 549
explicit, example of, 546
conversion operator. *See also* conversion function.
as constructor operator, 339
example of, 384, 385, 484, 568
for type relationship, 339
convertTo(), 496
copied-object pointer, *425*
to base not related to derived, 445
example of use, 461, 522
to interface, example of, 519
as member datum, 425
as return type, example of, 444
CopiedBuiltInPtr<T>, *426*, 427, 429, 430, 449
as public base, 430
CopiedObjPtr<Atom>, 447
CopiedObjPtr<T>, 425, 427, 429, 430, 431, 441, 444–449, 461, 462, 470
Coplien, J. O., 11, 173, 415, 446
copy
by assignment, 148
by initialization, 148
copy assignment operator, *96*
copy constructor, *96*, *148*, 149
argument initialization, 126
called, 149
example of, 98, 102
generated, 148, 191
for initializing function arguments, 100
for object initialization, 100
programmer-defined, 191
copy-assignment operator
example of, 99, 102
copy-on-write, *425*
copying
code for reuse, 271
arguments, avoiding with const references, 128
pointers, 191
cos(), example of use, 17
cost of change, 207, 265, 274
counted-object pointer
behavior of, 431
example of use, 437, 518
as member datum, 425
as return type, 439
counted-reference pointer
example of use, 524
CountedBuiltInPtr<Domain>, 519
CountedBuiltInPtr<T>, *433*
as public base, 436
CountedObjPtr<Element>, 439
CountedObjPtr<Node>, 439, 440
CountedObjPtr<T>, 425, 431, 434, *436*, 441, 445, 446, 448
coupling, avoiding unnecessary, 213, 599
Courant, R., 537
cout, 15, 27–29, 68, 72, 74
Cox, B., 6, 10
Coxeter, H. S. M., 508
creation type, *335*

CreationSize, *316*
Croz, J. D., 540, 559, 563
ctype.h, 25
CubicPolynomial, 111, 112
current(), 408
current-voltage curve. *See* IV (current-voltage) curve.
cursor. *See* iterator.
cyclic group, *509*

D

D, *167*, *262*, *515*
DAG (directed acyclic graph), *239*. *See also* class DAG.
Damasio, A. R., 3
Damasio, H., 3
damped least squares, *603*
damped SVD, 604
DampedRectSVDRep<T>, *604*
DampedSVDIteratedEquations<T>, 603, 605, *607*, 608, 611
Dangle, *191*, 192
dangling reference, *192*, 548
  with array reference objects, 387
  dangling reference problem, 193, 194
  exposed implementations and, 548
DATA, 42
data member, *86*, 89; *See also* member datum.
data model
  class, example of, 610
data modeling, 583, 584
data point, *584*
data structure, 3, 613
  changing, 207
  combined with functions is class, 4
  dynamic, 437
  recursion and, 60
  secret, 207
  traversal with pointer, 420
  variable size, 175
DataModel<T>, 587, 588, *589*, 605, 607, 610
  as public virtual base, 610
DAXPY, 136
deallocation, *176*
debugging, and inline functions, 56, 83
declaration, *38*, 114, *119*, 285
  as abstraction, 120
  argument, 122
  example (table), 38
  FORTRAN-compatible, 553
  forward, example of, 546
  friend member, 168
  function, 120, 121
  function pointer, 511
  function template, 136
  mandatory, 20
  placement in source code, 39, 75
  reading complex, 47
  variable, 19
  versus definition, 114, 119, 139
declaration statement, *13*, 75
  in for loop, 37
declarator, 38
decrement operator, 22
deep copy, *149*
default argument, *130*
  and copy constructor, 148
default constructor. *See* constructor—default.
#define, 76
definition, 114, *119*
  constructor, 146
  exactly one for extern objects, 179
  function template, 137, 138
  an implementation, 119
  static member function, 166
  versus declaration, 114, 119, 139
  virtual function, 237
degenerate parameters
  in nonlinear estimation, 603, 613
delegation, *279*, *415*; *See also* accessor.
delete, 31, 49, 71, 77, 78, 98, 165, 176, 188–194, 257, 420, 429
  andconst pointer, 189
  base class pointer, 234
  on function arguments, 193
  left side of assignment, example of, 427
  matched to new(), 190
  null pointer, 189
  pointer, 419
  on pointer not from new(), 420
  on pointer to base class, 257
delete [ ], 49, 78, 98, 190, 256, 420, 543
Demmel, J., 370, 451, 452, 467, 559, 579
DemoAtwoodsMachine, *495*
dereferencing
  intercepted, 424
  null pointer, 44
derivation, 280
  adding new derived classes, 343
  class scope and, 286
  and encapsulation, 300
  example of, 244, 247
  for extension versus commonality, 299
  from interface base class, 235, 251
  interface from interface, 400

derivation *(cont.)*
matching virtual function types, 237, 253
object factory, 255
public for inheritance, 373, 429, 439
See also inheritance.
selecting technique, 298
of template from template, 309
template problem, 447
derive, 271
derived class, *236*
derived publicly, *266*
derived type, 356; *See also* pointer—built-in—as parametric type.
derived virtually, *281*
DerivedUnits<T>, *498*
as public base, 498
design
bottom-up, 10
cost of, 218
evolution, 8
improvement, 199
object-oriented, 7, 10
of C++ language, 5
of overloaded names, 136
program, 7
reflected in FORTRAN subroutine names, 454
structured programming, 10
top-down, 10
destruction, *176*
of dynamic object, 189
of static object, 181
order of
automatic objects, 185
static local objects, 183
static nonlocal objects, 182
destructor, *96*, 98, *165*
example of, 98, 102, 256
not inherited, 268
at program termination, 182
virtual, 234, 443
example of, 315
Detlefs, D. L., 195
DGETRF(), 468
DGETRS(), 468
Dhatt, G., 218
DiagonalOfArray2d<T>, 416
diagram
base classes, 239
of variables, 69
See also object sketch, class DAG.
diamond-shaped class DAG, 296
dictionary
exercise, 537
paired arrays, example of, 529
Dietrich, Jr., W. C., 539, 577
differentiation, 590
dihedral group, *507*
Dijkstra, E. W., 10
Dimension, *315*, 347, 393
dimensional analysis, 493
dimensionality, array, *312*, *316*
DimensionedArrayShape<2>, 324
DimensionedArrayShape<ndim>, *323*, 324
as private base, 324
directed acyclic graph. *See* class DAG.
directed graph, *239*
disambiguation
of inherited names, 289, 297, 503
of overloaded functions, 155
dispatch, *345*
DispatchingList<T, AcceptorType>, 359
Distribute1<Op, Array>, 503
Distribute2<Op, Array>, *502*
DistributeS<Op, Array>, 503
distributing arithmetic
example of use, 564
DistributingAbelianGroup<Array, T>
as public base, 564
DistributingAbelianSemiGroup<Array, T>, *501*, 50
DistributingDivisionAlgebra<Array, T, S>, *503*, 504
as public base, 503
DistributingEquivalentCategory<Array>, 504
as public base, 503
DistributingField<Array, T>
as public base, 503
DistributingLeftScalars<Array, T, S>
as public base, 503
DistributingLinearSpace<V, S>, 580
DistributingSemiGroup<Array, T>, 502
division algebra, *476*
example of, 518
DivisionAlgebraCategory<V, S>, *493*, 504, 519, 5
as public base, 503, 518, 591
DO, 34, 41
do–while loop, 36, 37
documentation, 219
Domain, 484, 516, 519
domain and range
compared to arguments and return type,
dominance, 285, *288*
as generalized name hiding, 288
painless, 289
precedes access control, 290
proximity of base irrelevant, 289
with templates, 324

Dongarra, J. J., 136, 370, 451, 452, 467, 540, 559, 563, 579
dot operator. *See* member—selection operator.
dot(), 567
double, 22, 23, 134, 554
DOUBLE COMPLEX, 554
double dispatch, 343, *345*
DOUBLE PRECISION, 554; *See also* double.
double quote, 25, 50
downcast, *343*; *See also* type—cast.
DPPTRF(), 468
DPPTRS(), 468
Du Croz, J., 370, 451, 452, 467, 559, 579
dual-range voltage supply, 285
DualGaussianModel<T>, *610*
DualRangeVoltageSupply, 295, 296
Duff, I., 540, 559, 563
dynamic object lifetime, *176*
dynamic array, 47, 95, 188; *See also* array—formed (runtime sized).
dynamic binding, *327*
dynamic object, 188
  allocation/deallocation, 188
  construction, 188
  destruction, 189
  and exceptions, 108
  failure to delete, 193
  pointers to, 193

E

Eddy, F., 10, 261
Edelson, D. R., 446, 447
efficiency, 5, 7, 64, 65, 115, 128, 147, 180, 207, 261, 275, 276, 291, 292, 299
  array versus list, 441
  array, trade-offs, 312
  dynamic data, 168
  dynamic object, 189
  eliminating temporaries, 505
  by hiding base class name, 490
  and inline functions, 55, 82
  object size, 90
  passing functions by name, 488
  programmer time, 364
  of recursive functions, 60
  runtime versus space and design time, arrays, 313
  and side-effects, 125
  space, concrete array, 363
  template specialization for, 374
  templates to allow inlining, 505
  trade-offs, essential, 313
  virtual base class, 300
  virtual functions, 299
Eiffel, 6
Element, 208, *209*, 212, *213*, 214, 215, 217, 219, 437, *438*, 439, 440, 441
element (finite), *200*, 208, 211, 212
element table, *202*, 208, 209
element type, *312*; *See also* array—element.
Element*, 439
Elements, 216, *312*
ElementsOfMesh, 213, *215*, 216, 217, 437, *439*
Ellis, M. A., 5, 8, 9, 11, 62, 77, 115, 139, 190, 261, 262, 300, 356–358, 505, 527, 578, 628, 650, 651
else, 16
EltT, 465
encapsulation, 4, 87, 89, 97, 98, *207*, 218, 230, 300
  adjustment after derivation, 293
  advantages of, 211
  aids modularity, 291
  class, not object, 92
  of collection by iterator, 409
  compile-time security, 291
  costs of, 231
  of derivation, 295
  for derived classes too, 292, 300
  distinct from abstraction, 212, 218
  enabling abstraction, 233
  example of, 208, 227
  managing change, 263
  nested class, 110
  and reuse of implementation, 284
  sketch, 239
  as typing, 357
  using private constructor and friend, 463
  violated by public inheritance, 299
enclosing
  class, example of, 111
  class, of nested class, 110
  while loop, 37
end of file, 35, 74
endl, 16, 28, 68, 73, 83
enum. *See* enumeration.
enumeration, *25*, 70, 71, 134, 173
  conversion to int, 26
  conversions, 71
  example of use, 26, 285
  nested, example of use, 286
  specified value, 26
  as type, 26, 70
enumerator, 71
enumerator list, *71*
envelope class, *415*. *See also* accessor.

.EQ., 24
equality operator, 31, 45
  versus assignment, 23
EQUIVALENCE, 20, 21
EquivalentCategory<DerivedType>, *352*, 353, 354, 355, 360
  as public base, 353, 359, 591
errno, 552
errno.h, 552
error handling, 105; *See also* exception.
  C language to C++ exceptions, 541
  FORTRAN to C++ exceptions, 555
  and interface base classes, 257
  stream I/O, 173
  string conversions, example, 551
  string searches, example, 551
error message, 105, 258, 317
escape sequences, *24*
  table of, 25
evolution, program, 4, 205
exact match, function arguments, 133
example
  accumulation in double precision, 325
  array element copy, 383
  array element offset, 374
  Atwood's machine, 494
  axpy function template, 136, 138
  Boolean type, 151
  cartesian product, exercise, 509
  comma-separated output, 305
  complex numbers, 153, 166, 478
  dimensional analysis (units), 493
  dual-range voltage supply, 285
  dynamically created Gaussian function, 525
  file name manipulation, 544
  finite element mesh, 437
  fitting Legendre polynomials, 576
  Frobenius matrix norm, 407
  function evaluator, 526
  linear equations, 468
  linear fractional transformation, 480
  linked list, 168
  logarithm, 130
  mathematical expression evaluator, 518
  maximum element of array, 364
  mesh-reading program, 199, 203
  minimum element of array, 128, 131, 134
  modulo arithmetic, exercise, 505
  molecule of atoms, 442
  Newton's method, 36
  Newton-Raphson equations, 594
  nonlinear fitting, 583
  quaternion, exercise, 508
  random deviate, 123
  rational numbers, 194
  right-to-left binary method, 488
  searching intervals, 157
  string class, 542
  summing array elements, 129
  SVD for least squares, 575
  swapping function arguments, 127
exception, *105*, 115
  catching, 105
  causing memory leak, 423
  destructors run, 185
  example of, 106, 107, 111, 266, 585
  from C, 541
  from FORTRAN, 555
  handling, 107
  interface for, 257, 263
  not used in iostream, 173
  runtime type error, 460
  throwing, 105
exclusive or, bitwise, 27
exit(), 105, 182
EXIT_FAILURE, 60
EXIT_SUCCESS, 60
  example of, 211
experimental C++ features, 115
ExperimentSimulation, *252*, 254, 256, 263
explicit state represented as member data, 229
exponential notation
  on output, 28, 73
exponentiation, 22, 104
  operator not available, 160, 505
exposed implementation, 212, 219, 274
  example of, 375
  factored matrix representation, 469
  string class, 548
expression, *13*
expression tree, *523*
extension by public inheritance, *270*, 299
  for interfaced classes, 270
  in a template class, 309
extern, 53, 114, 121, 124, 179, 540, 563
extern "C". *See* linkage.
external composition law, *476*, 492
external linkage, *123*, *178*, 179, 194,
external object implementation, *539*
ExternalScalarsCategory<V, S>, *492*
  as public base, 493

F

\f, 25
Factored, *466*, *467*, *572*, *604*

FactoredLapackRect<T>, 431, 457, *458*, 460, 461, *462–464*, 466, 470
FactoredLapackSymPosDefPacked<T>, 457
factoring matrices, 452
factory, 253
Falkoff, A. D., 7, 535
Fallible<double>, 307
Fallible<int>, 307
Fallible<Interval>, 157
Fallible<Subscript>, 551
Fallible<T>, 157, *158*, 159, 173, 174, 307, *308*, 309, 327, 544
FallibleBase, *307*, 309
  as private base, 308
.FALSE., 24
FAMaker, 527, *529*, 532–537
fat interface, *346*, 358
field, *475*, 476, 478
FieldCategory<ComplexFloat>, 479
FieldCategory<T>, 478, *491*
  as public base, 479, 493
FieldScalarsCategory<V, S>, *493*
  as public base, 493
file
  header, 120
  implementation, 120
  input/output, 29, 75, 187
file name, 14
  string manipulation of, example, 544
file scope, *178*, *194*
  external linkage, 179
  internal linkage, 180
  replaced by class-scope objects, 179
finite element mesh, modifying, 437
finite element method, 199, 218
fitting line to points, 47, 95, 102
Flannery, B. P., 36, 47, 60, 313, 416, 537, 541, 571, 587, 595, 604, 611, 613–615
flexibility, array, 312
float, 15, 22, 23, 134, 554
float.h, 22, 94, 574
floating point number
  characteristics of, 22
  precision, 21
  style for constant, 23
FLT_MAX, 94, 169
Force, 495
form feed, 25
formal argument, *57*, *80*, 125
  reference, 81, 127
formed array, *313*
FormedArray1d<float>, 336, 337, 358
FormedArray1d<int>, 358
FormedArray1d<T>, 338, 589
  as private base, 588, 599
  as public base, 353
FormedArray2d<double>, 350
FormedArray2d<float>, 314, 325
FormedArray2d<T>, 314, *402*, 414, 416
FormedArray2d<VoltageSupply*>, 314
FormedPhysicalData<T>, *585*, 586, 611
FormedPhysicalDataBrowser, *586*
FORTRAN
  called from C++, 553
  calling C++ from, 555
  compared to C++, 3, 5, 6, 13
FORTRAN-77
  nonstandard extensions of, 6
FORTRAN-90, 6, 62, 218
FORTRAN analog
  array, 312, 553
  array classes, 414
  array storage format, 313
  assignment, 20
  BLAS subroutines, 136
  built-in types, 553
  CALL, 52
  CHARACTER, 50
  column 6, 13
  column-major array, 42
  COMMON block, 115, 124, 179, 207
  declaration order, 39
  DIMENSION, 41
  DO, 34, 37
  ENDIF, 32
  EQUIVALENCE, 20
  explicit type statement, 13
  exponentiation, 22
  FORMAT, 29, 73
  FUNCTION, 52
  function name, 122
  generic intrinsic functions, 89
  GOTO, 34, 37
  IF, 16, 32
  IMPLICIT NONE, 20
  index 1 begins array, 62
  INTEGER, 19, 21
  internal file, 30
  intrinsic function, 17
  LAPACK, 451
  leading dimension (LDA), 458
  list-directed READ, 15
  LOGICAL, 24
  loop index, 169
  mesh-reading program, 202
  multidimensional array, 41

FORTRAN analog *(cont.)*
PARAMETER, 39
pass by reference, 59
PROGRAM, 14, 60
READ, 62
REAL, 15, 21
RETURN, 52
subroutine, 13, 52
subscripting, 42
THEN, 32
type conversion, 21
variable-size array, 47
WRITE, 15, 62
FORTRAN array
element storage compatible with, 369
FORTRAN code
organization of, 115
FORTRAN compatible
array element storage, example of, 378
array pointers, 375
array projection, 389
FORTRAN subroutine
called from C++, example of use, 558
"FORTRAN", 553, 554
FortranArray1d<int>, 464
FortranArray2d<T>,414 , 461, 463, 464, 466
FortranSymmetricPackedSubscriptor, *561*
forward declaration, *87*
forwarding. *See* member function—forwarding.
forwarding function, *279*
fprintf(), 74
free format input, 200
free store, *188*
free(), *77*, *78*
freed, *176*
friend, 167–171, 215, 216, 218, 354, 458
class, 167
class, example of, 167, 169, 380, 384, 410
function, 167
function, example of, 426, 433, 485
member function, example of, 457, 462
nonmember function, example of, 352
Frobenius norm, 407
fscanf(), 68, 74
fstream.h, 30, 75
FTP address for source code, 9
function, *13*, 14, 52, 119
algebra with, 518
applied to collection, 533
argument. *See* argument.
body, *14*, 68
built-in pointer to, 511
C language called from C++, 540
call, 52
called by FORTRAN, 555
C++ called from C language, 541
called through pointer, 512
calling itself, 59
with class name, 91
client, 246
composition, 518
declaration, 115, *120*, 121
definition, 53, 115
inline, 55, 82
syntax, 52
delayed evaluation, 514, 518
FORTRAN-compatible, declaration of, 553
with hidden parameters, 515
identifier, *121*, 122
name, overloaded123
nonmember, 114, 123
non-virtual, 275
overloading, 88
pointer to*See* function pointer.
prototype, *53*, 55, 64, 67, 82, 87, 114
recursion, 59
return
initialization and, 131
type conversion, 131
without arguments, 131
function argument. *See* argument.
function argument matching. *See* argument matching.
function call
ambiguous, 133, 135
and const member, 144
inline, 55, 82
no match, 133
overhead of, 55, 82
function call operator, 31, 164
for string class, 543
function call through interface, *See* virtual—function.
function domain
arguments as, 122
function evaluator, example, 526
function object, *511*
example of, 598
function overloading, 132
function pointer, *511*
declaration, 511
initialized with template, 512
as member datum, example of, 517
function range
return type as, 122
function structure, *351*

function template, *104*, 136, *304*
arguments must include parameters, 321
client of array interface, example of, 320
as client of class template, 348
declaration, *136*
definition of, *137*
definition versus declaration, 104
example of, 104, 305
example of use, 305
15 template parameters, example of, 499
forward declaration, example of, 320
function pointer initialized with, 512
matching, example of, 105
for multicomponent arithmetic, example, 601
nontype arguments, 505
nontype parameter arithmetic, 499
parameters and argument types, 488
parameters not in arguments, 505
simulated with class template, 488
too powerful, 325, 351
using common names, 306
versus template function, 304
function type, *121*, 122
function-structure category, *354*, 358, 478
Function<Domain, Range>, *517*
as private base, 517
functional programming, 139
Functional<Domain, Domain>, 536
Functional<Domain, Range>, *516*, 517
as public virtual base, 516, 517, 589
FunctionalAlgebra<Domain>, *518*–527, 533, 535
FunctionalAlgebra<double>, 529, 531, 532
FunctionUndefined, *529*
fundamental type. *See* built-in—type.
FundamentalUnits<T>, *498*
as public base, 498

G

Gamma, E., 262
garbage, *193*
array, 387
collection, 195, 446
Garbow, B. S., 451
Gaussian error distribution, 587
Gaussian function, 525, 610
GaussianFunctional<Domain>, *525*
Gautron, P., 139
GBIBComponent, 341
.GE., 24
General Purpose Interface Bus (GPIB), *226*
generality. *See* abstraction.
generated
assignment operator, 163, 164, 268
constructor, 268
copy constructor, 148, 149, 276
default constructor, 148
destructor, 268
member functions, and base classes, 269
generic datatype, *327*
geometrical programming, 116
geometry
classes for, 85
get/set functions, 92, 230
getopt(), 60
Ghezzi, C., *139*, 415
glitch, in experimental error, 587
global function, 155; *See also* nonmember function.
conversion operator, 161
as friend of class, 167
name conflict, 173
operator, 161, 162
and side effects, 125
static member function as alternative for, 166
global scope, *194*
global variable, 124
Goldberg, A., 6, 261, 415
Golub, G. H., 417, 456, 470, 571
Gorlen, K. E., *11*, 195, *299*, 415, *446*, 594, 602, 613, 613
GOTO, 34
GPIB, *226*, 231
address, 227
classes for, 260
controller simulator, 226, 250
example, 270
simulator, exercise, 263
GPIBComponent, 340, 341, 343, *344*, 345, 346, 359
as public virtual base, 344
GPIBComponentAcceptor, *344*, 345, 346
as public base, 345
GPIBComponentPrinter, *345*
GPIBComponent_TC, *341*, 342, 343
as public base, 342
GPIBController, 227, 228, *246*, 247, 248, *251*, 253, 277
as public base, 247, 254
GPIBController_GC, *247*, *341*, 359
GPIBController_GC_GC, *341*, 359
GPIBController_GIS, *254*, 256
GPIBController_Stub, *226*, 227, 228, 238, 246, 247
GPIBInstrument, 243, *244*–246, 250, 263, 271, 277–284, 287, 289, 292, 298, 301, 341, 352, 535
as public base, 244, 247, 278, 294

GPIBInstrument *(cont.)*
as public virtual base, 280, 281, 285, 293, 342, 344
GPIBInstrument*, 535
GPIBInstrumentData, *277–281*, 289, 298, 299, 301
GPIBInstrumentData_GI, *280–285*, 289, 294
as private base, 281, 293
GPIBInstrumentSimulation, *251*, 253, 255, 263, 420
as public base, 251, 252
GPIBListener, 263
GPIBTalker, 263
Gracer, F., 539, 577
gradient vector
from Taylor expansion, 588
Graham, R. L., 59
grammar
for simple function evaluator, 527
Greenbaum, A., 370, 451, 452, 467, 559, 579
Griewank, A., 589
group, *475*
group theory, 473
GroupCategory<T>, *489*, 490
as public base, 491
.GT., 24
guideline. *See also* important rule, style, warning.
argument types, 129
assignment operators in interface base classes, 400
asymmetric binary operators, 162
avoid casts, 343
avoid conversions that breach encapsulation, 549
avoid extern file-scope objects, 179
avoid fixed maximum sizes, 553
avoid global side effects, 125
avoid hiding, 275
avoid interface base class constructors, 241
avoid local static objects, 180
avoid malloc() and free(), 78
avoid protected, 292
avoid replicated code, 265
beware of pointers to temporaries, 549
built-in pointers as member data, 192
checked types for search functions, 159
clone function return type, 443
conventional operator definitions, 161
coordinate template specializations, 349
declare base class functions virtual, 275
deep interface, shallow implementation DAGs, 297
default arguments in declarations, 131
define interface functions as constraints, 242
do loop, 36
don't mix FORTRAN and C++ I/O, 556
encapsulate legacy code, 548
extern linkage, 124
extern usage, 179
friend declarations, 171
function call operator use, 164
function templates with template parameter arguments, 325
global symmetric binary operators, 162
good concrete class, 414
group error classes into interface categories, 257
group private data and functions into a class, 291
hide data structures, 207
hide implementation arrays, 212
hide malleable information, 207
implement virtual interfaces with private bases, 284
initialization of static objects, 184
inline sparingly, 55, 82
interface base class virtual destructor, 234
interface bases should be abstract bases, 243
match class design to legacy library design, 539
meaningful class names, 331
modified measure-before-inline approach, 56, 83
name abstractions, 208
nested classes, 113
no data in interface base classes, 242
no delete on arguments, 193
no pointers to automatic variables, 193
non-const reference argument, 128
one conversion per pair of classes, 153
overload, don't encode, 559
pair browsers and iterators, 410
pair new() and delete calls, 194
passing functions by name, 488
private member data, 89
provide default constructors, 189, 269
pure side-effects in classes, 125
reading declarations, 47
relate related operators, 163
relate template specializations, 349
replace built-in arrays with array classes, 190
return by reference, 132
return type for assignment operators, 164
signal input modification side effects, 125
simulating function templates, 488
throw objects; catch interfaces, 257
use 0 not NULL, 77
use abstract base classes, 276

use class scope names, 180
use const arguments for assignment operators, 164
use const arguments for copy constructors, 150
use const members, 145
use const not #define, 76
use derivation, 298
use interface base classes, 291
use iostream.h, 542
use iterators for sequential elements, 409
use obvious overloading, 136
use parameter-invariant base classes for templates, 309
use private member data, 230
use pure virtual functions in interfaces, 243
use standard functions, 61
use standard initializer order, 269
use stream I/O, 72
uses for -> operator, 165
while loop application, 35

H

Habermann, A. N., 6
Halbert, D. C., 218
Hall, C. A., 218
Halliday, D., 494
Hammarling, S., 370, 451, 452, 467, 540, 559, 563, 579
handler
exception, 107
Hanson, R. J., 136, 540, 559, 563, 570, 571, 580, 603, 604, 606, 613
Harbison, S. P., 25, 62, 260, 578, 579
header file, *15*, 88, 113, 114, *120*; *See also* standard—header file.
heap, 77, *188*
heap object
cannot be shared with copied-object pointer, 430
failure to delete, 193
to initialize pointer object, 446
managed with pointer class object, 425
hello world program, 67
Helm, R., 262
heterogeneous collection. *See* collection—of base class pointers.
hexadecimal, 25, 26
hides, *267*
hiding, 297
in algebraic structure categories, 482
block scope compared to class scope, 287
to provide more efficient function, 490
versus overloading, 288
hierarchy, *261*
Hilbert, D., 537
horizontal tab, 25
Horner's method, 64, 84
Horstmann, C. S., 11
Hydrogen, *442*, 445

I

I/O, 15, 27
buffer flushing, 68
to file, example, 187
errors, 74
mixed C and C++, 542
mixed FORTRAN and C++, 556
setting const object from, 39
stream, example of use, 14, 56, 305
streams versus stdio, 577
type safe, 542
unformatted, 62
IAND, 27
identifier resolution, 285
identity, 475
IEEE-488, 226
IEOR, 27
IF, 16, 32, 34
if statement, 16, 32
if–else statement, 33
ifstream, 187, 188
Ikebe, Y., 451
implementation, *335*
commonality, 100
captured in base class, 278
inheritance versus templates, 306
definition, as, 119
dependency, 22, 116, 188
exposed, 212, 219
file, 113, *120*
impact on design, 8
member function, 143
by public inheritance, 299
implementation base, *266*
implementation base class, *266*
composed with interface base, 294
example of, 277
for template class, 307
templatized, 352
versus interface base, 271
implementation category, *298*
compared to interface category, 298
implementation of interface

implementation of interface *(cont.)*
with derivation, 280
implementation reuse
templates versus inheritance, 326
implementation structure commonality, *303*, *306*
IMPLICIT, 20
IMPLICIT NONE, 20
implicit state
external to computer, 229
important rule
check errno after C library functions, 552
delete, and null pointer, 98
function declaration, 119
include iostream.h, 15
include iostream.h for I/O., 68
initialize pointer objects with new, 435
insert your whitespace, 28, 73
the main() function, 14
match delete and new, 190
new directly into pointer object, 430
override clone function, 443
private bases are encapsulated, 282
private bases are not interfaces, 282
public implementation is not encapsulated, 274
put function prototypes in header files, 54
semicolon ends class definition, 86
terminating template recursion, 391
unhide overloaded virtual functions, 288
#include, 15, 53, 54, 120
inclusive or, 27
increment operator, 22
and loops, 37
indefinite matrix, *452*
indentation, 32, 61, 83
indirection, *43*
intercepted, 424
supporting object views, 333
indirection operator, 45, 427, 431
optional, for function call, 512
information hiding, *207*, 218, 230
and implementation inheritance, 274
mediating objects, 214
for module partitioning, 218
partial, 215
state representation, 230
violating, 212, 548
inheritance, *4*, 267, 298, 300; *See also* derivation.
in function-structure category, 483
mechanics of, 284
and names in class scope, 286
protected, example of, 377
public, example of, 412
pure virtual function, 261
from template, 309
from template class, example of, 393
versus hiding, 287
inheritance hierarchy, *261*; *See also* class DAG.
inherited, *287*
initialization, 148, *175*
array, 41
of array of pointers, example of, 254
base class reference, 334
base class reference function argument, 334
base subobject, example of, 271
const reference, 332
constructor calls, 268
default, 184, 195
of function argument, 81
function pointer, overloading and, 512
and function return, 131
multi-dimensional array, 42
object, 145
order of, 184, 194, 196
for base subobjects, 268
for members, 147, 268
of reference, 51, 79
of reference member datum, 248
example of, 249, 254
of static local variable, 180
static member datum, 94
initializer, 38, 39
expressions, 147
object (constructor), 145
initializer list, 41
inline, 55, 56, 82, 83, 123, 231, 299, 486
function template, example of, 159
member function, example of, 393, 410
and templates, 409
virtual function, 299
inline function
to avoid template expansion, 324
in class body, example of, 375
and compilation time, 55, 82
overuse of, 55, 82
and template versus function pointer, 502
inner product, 614
Inner1, *110*
Inner2, *110*
input, 15, 68, 88, 106
binary data, 62
constructor, example of, 585
from file, 30, 75
fixed-column, 28
free format, 200
input modification side-effects, *124*

input operator. *See* operator<<().
input-modification side-effect
array argument, 130
controlling, 125
input/output, 27; *See also* iostream.
operators for, 171
instance, *86*, *101*; *See also* object.
class, 86
creation of, 88
derived class, 266
instrument control objects, 225
int, 22, 23, 134, 554
from char, 24
INTEGER, 21, 554
integer constant
hexadecimal notation, 26
octal notation, 26
INTEGER*2, 22, 554
INTEGER*4, 22, 554
INTEGER*8, 22
integral type
char, 70
enumeration, 71
interface base, *234*
interface base class, *234*, *236*, 242, 262, 334
access through pointer to, 420
addition to, for extension, 295
aggregation of objects through, 441
array, dimension-independent, 318
array, element type independent, 318
client class, 250
client function, 250
code size advantage, 409
composed with implementation base, 294
const layer, example of, 398
example of, 243, 246, 251, 257, 315
for exceptions, 257
functions, defined, 242
just a class, 242
as member datum type, example of, 403
member function definition, example of, 316
no data, 242
no initializer needed, 284
non-const array layer, 400
number of specifications versus applicability, 398
programs without, 327
reference to, example of, 247
sketch for, 238
as template parameter, 326
templatized, example of, 584, 585, 589, 595, 607
versus abstract base class, 243
versus template, 326
versus template for commonality, 397
interface category, *231*, *234*, 260
classes are not concrete, 270
compared to implementation category, 298
example of, 231
interface type, *335*
interfaced array, *363*
interfaced array class, 313
interfaced function pointer, 517
InterfacedArray2d<A>, 417
InterfacedArray2d<T>, *401*, 471
as public base, 402
InterfacedArrayProjection1d< ConcreteFormedArray2d<double> >, 406
InterfacedArrayProjection1d<ConcreteArray2d>, *405*, 406
internal binary composition law, *474*
internal file, 30
internal linkage, *123*, *178*
uses for, 124
Interval, *157*, 171, 174
InterViews, 262
inverse, 474
inverse(), 483, 490
invert(), 480, 489, 490
_IOLBF, 83
iomanip.h, 29, 74
IOR, 27
iostream, 27, 62, 171, 173, 542, 579
iostream.h, 15, 29, 68, 74, 88
<iostream.h, 542
irresponsible sharing, *445*
is-usable-as
array interface, 313
commonality, 237, 260
public derivation and, 271
relation, public interface base class, 280
transitive, 271
isdigit(), 25
ISHFT, 27
isNull(), 427, 433
IsoFunction<Domain>, *517*
IsoFunctional, *516*
IsoFunctional<Domain>, *516*–522, 536
as public virtual base, 517, 518, 521, 522, 523, 524, 525
isomorphic classes. *See* template.
ispunct(), 25
istream, 172
isupper(), 25
IteratedEquations<T>, *595*–598, 600
as public virtual base, 599, 607

iteration, *406*, 407
built-in C++, example of, 407
design, 8
with iterators, 408
in nonlinear estimation, 595
for nonlinear least squares, 587
iterator, *169*, 213, *364*, *406*, *547*
example of, 215, 216, 596
example of use, 216, 217, 533, 597
lexer as, 527
mapping, exercise, 536
read-only. *See* browser.
specialized, example of, 439
for strings, example of, 546
IteratorType, 409
IV (current-voltage) curve, 248
Iverson, K. E., 7, 535
IVTester, 248, *249*, 250, 256, 397

J

Jacobian
partials for Newton-Raphson, example of, 600
Jazayeri, M., 139, 415
Jenks, R. D., 261, 327, 357
Jerrell, M. E., 613
Johnson, R., 262
Juedes, D., 589

K

K&R C, 67
kaleidoscope, 508
Kappraff, J., 507, 508
Kennedy, B. M., 415, 447
Kernighan, B., 67
keyword, for complete list see [*The Annotated C++ Reference Manual*, Section 2.4]; 13, 152
Kincaid, D. R., 136, 540, 559, 563
kits, *262*
Klema, V. C., 451
Knowns1d, 466
Knowns2d, 466
Knuth, D. E., 10, 59, 488
Knuttila, K., 139
Koenig, A., 7
Krogh, F. T., 136, 540, 559, 563

L

language
and abstraction, 3
LAPACK, 9
declarations for, 553
design dimensions, 454
organization of, 451
potential error, 454
subroutine arguments, 455
symmetric array storage example, 560
LapackFactored<Rep>, 464, *465*, 466, 469
LapackMatrix, 468
LapackRect<T>, 431, *457*, 458, 460, *461*, 463, 464, 465
LapackSubroutines, *557*
LapackSymPosDefPacked<T>, *457*
LapackUnfactored< RectLURep<EltT> >, 468
LapackUnfactored< RectSVDRep<double> >, 605, 607
LapackUnfactored<Rep>, *464*, 465, 468, 469, 470, 606
Laplacian smoothing, *220*
largest integer, 22
lattice
versus DAG, 261
Lawson, C. L., 136, 540, 559, 563, 570, 571, 580, 603, 604, 606, 613
LDA. *See* array—stride, leading dimension.
.LE., 24
leading dimension (LDA), 469
Leaker, *423*, 424, 430
least-squared error, *587*
LeastSquares<A>, *605*, 606, 608
left singular vectors, *570*
left-shift operator. *See* <<.
LeftScalarsCategory<V, S>, *493*
as public base, 493
legacy, *539*
Legendre polynomial, *537*, 581
fitting, example, 576
Lejter, M., 173
Levenberg-Marquardt nonlinear least squares, 595, *613*
Lexer, *527*, 529, 536
lexical analyzer, *527*
library
for C++, 5
class versus subroutine, 577
FORTRAN, called from C++, 553
subroutine, 539
lifetime, *176*, 192
limits, numerical, 22, 574
limits.h, 22
Line, 85, 86, *87*, 89, *90*, 91–94, 115–117, 173
linear algebra, *476*
linear equation, 451, 452
linear equations. *See also* matrix.

LAPACK for, 451
solution with C++, example of, 467
from Taylor expansion, 588, 595
linear fractional transformation, 480
linear least squares
solved by pseudoinverse, 570
linear space, *476*
algebraic category, example of, 496
LinearFractionalTrans, *480*
LinearizationIterator<T>, *596–600*, 603, 610
LinearSpaceCategory<V, S>, 614
as public base, 497
Lines, L. R., 603
LineSegment, 145, 173
link, *168*
linkage, 194
"C", 540
"C++", 540
default, 124
external, 123
and file scope, 178
"FORTRAN", 553
general case, 577
internal, 123
linkage rules. *See* [*The Annotated C++ Reference Manual*, Section 3.3].
linkage specifier, *121*
linked list, *168*
example of use, 437
linker, 578
versus linkage, 578
LINPACK, 136
Linton, M. A., 262
Lippman, S. B., 11, 139, 189
Liskov, B., 357
LISP, 195, 535
List<GPIBComponent*>, 341
List<ParserFunctionEntry>, 537
List<T>, 169, *170*, 171, 174, 219, 341, 437
ListIterator<T>, 169, *170*, 171, 174, 439
as public base, 439
Loan, C. F. V., 7
local scope, *56*, *178*, 194
local variable, *56*, 57, 80, 180
allocation, 176
function argument, 57
and recursion, 59
returned by reference, 132
static, 180
this, 92
locale, *579*
logarithm, example, 130
logical operator, 31
table of, 24
logical test, 33
logical values, 24
long, 22
long double, 22, 23, 134
long int, 22, 23, 554
loop, 34, 36, 37
comparison table, 35
example of use, 41, 169, 270
and inline functions, 55, 82
Lorensen, W., 10, 261
lowercase, 19
.LT., 24
lvalue, *367*

M

main(), 14, 60, 88, 113, 182
example of, 60, 87, 107, 211, 611
main program required, 14
maintainability, 265, 276
malloc(), 77, 78
manipulator, *16*, 28, 29, 35, *68*, 73, 74
many-to-one relation
pointer class for, 431
mapping
functions as mathematical, 123
MappingBrowser<Domain>, 536
Marquardt, D. W., 603
Martin, B., 300
Mass, 495, 499
match with promotion, *134*
math library, 17
math.h, 17
Mathematica, 535
compared to C++, 7
mathematical
expression built at runtime, 518
routine as abstraction, 206
mathematics analog
function name, 122
function type, 122
and functional programming, 139
operator notation, 159
Matlab
compared to C++, 7
Matrix, 330, *331*
matrix
as algebraic structure, 476
augmented, class for, 605
banded, 452
with BLAS implementation, 563
class from FORTRAN subroutines, 457

matrix *(cont.)*
eigenvectors, 470
in-place factorization, 460
indefinite, 452
inverse, 570
LAPACK storage methods, 452
LU factoring, example, 557
multiplication, 41
positive definite, 452
projection, example, 566
structure, *452*
symmetric, 452, 457
symmetric positive definite packed, 467
tridiagonal, 452
max(), 364, 365
maxArray2dElement(), 366, 415
maximum likelihood, *587*
maximum value
floating point (FLT_MAX), 94
McKenney, A., 370, 451, 452, 467, 559, 579
member, *86*
access to, 87
data, 86, 89
function, 86
hiding, 267
initializer, 146
of nested class, example of, 112
private, 208
public, 208
selection operator, 31, 92, 430
example of, 429, 436
pointer class and, 426
member datum
as abstraction, 92
as state, 229
access, 92
built-in pointer, example of, 376, 426, 546
const pointer, example of, 382
const reference, example of, 412
copied-object pointer as, 425
counted-object pointer as, 425
counted-object pointer, example of, 524
declaration, 89
direct access to, 231
function pointer as, 517
inside member function, 92
initializer, *146*
needed, 148
not needed, 147
interface base class type, example of, 403
none in interface base class, 236
parameterized, 322
pointer to const object, example of, 380, 410
pointer, for referential aggregation, 421
protected, example of, 426
reference, example of, 596
static, 94
template parameter as type of, example, 401, 523, 592
member function, *86*
ambiguous inheritance of, 283
base class called from derived class, 269
base class, definition of, 278
as behavior, 229
call, 88
through interface, example of, 292
from another member function, example, 150
base class member function, 267
class template, 102, 310
const, 144
as constraint, 275
to control objects, 231
coordinating for memory management, 100
derived class, 267
design of, 100
destructor, 96
determining complete set of, 297
example of, 228, 244
exercise, 263
forwarding, 265, 279, 299
to member pointer, example of, 403
example of, 278
instead of multiple derivation, 300
problems, 279
to wrap arrays in interfaces, 401
function template, example of, 310
implementation, 91
independent of template parameter, 307
not used in base class, 377
operator, 96, 161
overloaded, 88, 143
overriding virtual, example of, 272
pointer to, 513
private, 98
example of use, 530
role of, 291
to prevent use, 400
purpose, 143
reasons for declaring, 297
restrictions, 100
special, 96
static, 95
example of, 557
user-must-define, 354
working on two objects, 92

member variable. *See* member datum.
memory
  address, 43
  allocation bug, 193
  leak
    exceptions and built-in pointers, 423
    prevented with pointer class, 430
  management, 100, *175*
    ANSI C string functions and, 542
    in extended example, 437
    reference counting, 446
  object, 68
menu system as abstraction of control flow, 535
mesh, *200*, 208, *209*, 210–212, *213*, 214–218, 220, *437*, 439, 440
  quality, 200
  refinement, 219
  representation, 200
    adding new features to, 206
  structured, 219
mesh-reading program, 203
  critique of
    FORTRAN-like, 205
    with encapsulation, 211
    with information hiding, 217
Meshkat, S., 200
Metcalf, M., 6, 62, 218, 415
MetricSpaceCategory<V, M>, 614
  as public base, 591
Meyer, B., 6, 10
Meyers, S., 11, 173, 275, 299, 300
Milenkovic, V., 196
minimum array element, example, 128, 131
minimum norm solution to linear least squares, 570
mix-in, *358*
C mixed with C++, 76, 80, 83
MOD, 23
model, 583, *587*
Modula-3
  separation of interface and implementation, 260
modulus operator. *See* %.
Molecule, 442, *444*, 445
Moler, C. B., 136, 451
monoid, *475*
MonoidCategory<T>, *488*, 505
  as public base, 489, 491
more(), 408
multicomponent number, 589; *See also* complex numbers, example—quaternions, exercise.
multidimensional array, 41
multiple declarations, 120
multiple derivation, 243, *300*
  in base class composition, 294
  class DAG sketch, 245
  in concrete class, example of, 390
  and diamond-shaped class DAG, 296
  dominance for name resolution, 288
  example of, 244, 278, 281, 285, 491
  gives class DAG, 261
  to implement interface, 283
  object sketch, 245
  and virtual bases, 300
multiple inheritance. *See* multiple derivation.
multiple interfaces. *See* multiple derivation.
multiple precision, templates for, 100
multiplicative algebraic structure, *475*
Murray, R. B., 173
MySimulators, *255*
M_E (constant e), 130
M_PI (constant π), 17

N

\n, 25
Nackman, L. R., 195, 200, 300, 312, 539, 577
name, 38, *284*
  access declaration for, 294
  case, 38
  collision or conflict, 112, 173, 194, *289*
    accidental, 289
    avoiding, 167, 495
    at file scope, 179
  commonality, *303*, *305*
    provided, example of, 365
  dominant or unambiguous, 288
  of function converted to pointer, 511
  inheritance of, 284
  length, 38
  mangling, and linkage, 578
  subroutine convention, LAPACK, 452
name-space pollution, *194*
naming convention, 14
  replaced by function overloading, 557
.NE., 24
negate(), 480
  and algebraic structure, 483
NegativeSize, *316*
nested
  class, *110*
    example of, 110, 170, 266, 307, 316, 547, 604
  enum, example of, 341, 527
  typedef, example of, 373

new, 31, 47, 49, 71, 77, 78, 97, 176, 188, 189, 190, 191, 194, 420, 427, 431
  called in member initializer, example of, 434, 461
  called to initialize base class, example of, 520
  to initialize pointer object, 445, 446
  matched to delete, 190
  and object copy, example of, 427
  overloading, 189
  and pointer classes, 430
  pointer returned by, 419
new [ ], 78, 189
newline, 25, 68, 83
Newton's method, 36
Newton-Raphson algorithm, example, 594
NewtonRaphsonFunction<RallT>, *598*, 599, 600
NewtonRaphsonIteratedEquations<RallT>, 599, 600–605
Nierstrasz, O., 357
NoDangle, *192*, 421–423
node, *168*, 169, *170*, *200*, 208–215, 220, 437, 438, 440
node table, *202*, 208, 209
NodeReader, 213, *214*, 215, 438, *439*, 440
NodesOfElement, *216*, 219, 437, 439
NodesOfMesh, 213, 217
noise, impact on nonlinear estimation, 603, 613
NonLeaker, *431*
nonlinear equations, class for, 607
nonlinear estimation
  with curvature matrix, 613
  via stochastic inverse, 613
nonlinear least squares, 583
  iterative solution of, 609
nonmember function
  client of interface, example of, 319
  exercise, 263
  maintainable and extensible, 237
  template, 325
    example of, 351, 383
  using virtual functions, example of, 243
NonNegative<int>, 482, 488
NonNegative<T>, 483, 484
nonterminal, *527*
nonzero is true, 24
NoRealRoots, *111*, 112
norm, *614*
normal equations, *613*
normal error distribution, 587
normal probability density function, *515*
NormalDensityFunction<Domain>, *515*, 536
NormalizedLapackFactored<Rep>, 581
NormalizedLapackUnfactored<Rep>, 581
.NOT., 24, 27
nrerror(), 541
NULL, 76, 83; *See also* null pointer.
null character, *50*
null pointer, *44*, 76
  delete ok, 189
  delete on, example of, 256
  and integer 0, 76
  returned by new, 77
  testing for, 429
null termination
  maintaining in string class, 549
  string model, 547
Number, 89, 92, 94
numeric types
  comparison (table), 22
Numerical Recipes in C, 313
  calling C++ function, 541
NumericalLimits<T>, *574*, 599
NumericalRecipesErr, *541*

O

O'Brien, P. D., 218
object, *4*, 7, *18*, *68*, 352; *See also* instance.
  allocation, 175
  automatic, 176, 185
  cleanup, 176
  creation, 113
    example of, 240, 256
    preventing, 234, 376
    specification required for, 334
    versus use, 262
  deallocation, 176
  declaration of, 86
  design versus use, 230
  destruction, 113
  dynamic, 176, 188
  factory, 253, 262
    example use of, 254
    implementation, 255
    interface to, example, 253
  file scope
    external linkage, 179
    internal linkage, 180
  initialization, 88, 175
    as conversion, 151
    member datum, 147
    via constructor, 145
  life cycle, 175, 268
  lifetime
    and aggregation, 421
    control via accessor, 404

controlled by pointer, 419
controlled, in containing aggregation, 421
controlled, in referential aggregation, 421
passed to another aggregate, example of, 462
programmer-defined reference, 388
releasing control of, 429
safe with pointer class, 430
shared, pointer class for, 431
sharing, in containing aggregation, 422
sharing, in referential aggregation, 423
string storage, 549
local scope, static lifetime, 180
post cleanup, 176
preinitialization, 175
physical versus programming, 225
programming, 227
to represent function, 511
as representation, 227
required with member pointer, 513
size, 90
static, 176, 178, 180
temporary, 339
type, 332
impact on function called, 275
type versus reference type, 334
use
through base class pointer, 241
versus design, 230
versus creation, 240
versus interfaces, 398
object file, 119
object-oriented
example, 273
not, 275
programming, *4*, 7, 85, 225, 227, 259
benefits of, 4
terminology, 10
object sketch, 238
array reference, 386
base classes, 239
Booch, Rumbaugh, etc., 261
concrete array (no interfaces), 371
concrete array projection, 395
duplicate base classes, 283
encapsulation, 239
exercise, 263
extension of interfaced class, 271
factored LAPACK represention, 462
inheritance, 271
interface base class, 238
interfaced base class, 281
linearization iterator, 597
member function forwarding, 279
multiple derivation, 245
nonlinear equations, 608
pointer, 43
private base with virtual interface, 282
public base class, 266
reference member datum, 248
unfactored LAPACK represention, 461
Objective-C
compared to C++, 6
octal, 21, 25
offset
packed array example, 560
offset(), 370
ofstream, 187, 188
one-to-one relation
pointer class for, 425
Op, 502
operation, *21*
operator, *96*, *160*. *See also* subsequent entries for specific operators.
arithmetic (table of), 23
arity fixed, 160
assignment, 26
associativity, 30, 31
bitwise, 26
class member access, 165
comma, 42
conversion, 173
defined via algebraic structure categories, 473
equality, == and !=, 45
explicit call, 160
for exponentiation (none), 160, 505
function call, 164, 173
explicit, 160
as function, example of, 513
and functions, 160
global, conversion, 161
global, restriction on, 161
indirection, 45
input/output, 27, 171
logical, 24
member function, 96, 161
member-selection, 92
new, 47
not, 24
number of arguments, 161
overloading
–> and *, 427
precedence, 30, 31
projection, 165
relation between related, 161
relational, 23, 45

operator *(cont.)*
relational (table of), 24
symmetric binary, 486
symmetric, as constructor, 550
unary, 161
undefinable, 160
user-defined input, 172
user-defined output, 171
for user-defined types, 159
when to provide, 159
operator constructor, *551*, 578
operator!()
example of, 151
operator!=(), 351, 433, 545
operator()(), 164, 319, 514, 516
for array subscripting, 318
example of,377 , 515, 516, 545, 598
function call operator, 514
overloaded, example of, 519
operator*()
example of, 426, 433, 443, 568
operator*=(), 479
and algebraic structure, 481
example of, 486, 492
not in place for matrices, 565
for semigroup category, 485
operator+()
and algebraic structure, 483
example of, 545, 550
for string class, 543
operator++(), 22, 165
example of, 491
operator+=(), 479
and algebraic structure, 483
example of, 522, 545, 550
operator–()
and algebraic structure, 483
operator––(), 22, 165
example of, 491
operator–=(), 479, 489
and algebraic structure, 483
example of use, 36
operator–>(), 165, 446
and built-in type, 446
example of, 429, 436, 443
operator/()
and algebraic structure, 481
example of, 493
operator/=(), 479, 489
operator<()
example of, 545
operator<<(), 171
example of, 14, 87, 552, 598
operator<=()
example of, 545
operator=(), 268
operator==()
example of, 433, 545
operator>()
example of, 545
operator>=()
example of, 545
operator>>(), 27, 28, 72
example of, 14, 209, 210, 214, 215, 258, 552
operator[ ](), 49, 306
array projection, dimension 0, 367
example of, 98, 209, 545
restricted to one argument, 318
.OR., 24
Orange, *359*
Orlow, S. M., 11, 195, 299, 415, 446, 594, 602, 613
orthogonal, 313
ostream, 171
Ostrouchov, S., 370, 451, 452, 467, 559, 579
ostrstream, 579
out of line, opposite of inline
Outer, *110*
output, 15, 68, 88, 91
to file, 29, 75
fixed-column, 28, 73
formatted, 29, 74
operator. *See* operator<<().
overloaded, *123*
function, compared to function template, 136
functions, and pointers, 512
virtual function, example of use, 285
overloading, 88, *123*, 132
of constructor, 145
and friend functions, 167
function call operator, 164
member function, 143
in place of naming convention, 557
precedes access control, 290
resolution, example of, 91
versus hiding, 287
override, *273*
overriding virtual functions, 287, 297; *See also* virtual function.
names, 287
problems with forwarding, 279
Oxygen, 442, 445

P

packed storage

symmetric and banded matrices, 559
paging, 55, 82
Pakin, S., 7, 535
PARAMETER, 39
ParameterFunctional<Domain>, *524*
parameterized type, *101*, *330*
base class, by derived class, 355
built-in, 330, 331, 356
const, 332
template, 330
parameterized function. *See* function template.
parametric polymorphism, *327*
parentheses
and call through member function pointer, 513
and call through function pointer, 511
for function call operator, 164
for readability, 30
Parnas, D. L., 10, 207, 218
ParserFunctionEntry, 537
parsing, *527*
partial derivative
automatic differentiation, 589
from Taylor expansion of error, 588
partial differential equations, 199
PASCAL
compared to C++, 6
pass by reference, *59*, *127*
without pointers, 79
pass by value, *58*, *81*, *126*
copying, 148
problem for clients of base classes, 276
Patashnik, O., 59
peak fitting, 583
performance measurement
and inline functions, 56, 83
Perry, D. E., 6
physical constant, 39, 75, 179
physical objects, 225
Physical<T, m, l, t, q, k, i, a>, *497*
PhysicalConstants, *180*
PhysicalData<T>, *585*, 605, 607
as public virtual base, 585
PhysicalDataBrowser<T>, 584, 585
as public virtual base, 586
pipe, 27, 71
pivot, *470*
planning for change, 87, 89, 94, 299, 306, 315, 343
Plauger, P. J., 578, 579
Plexico, P. S., 11, 195, 299, 415, 446, 594, 602, 613
Point, 85, *86*, 87–93, 115–117, 143, 144–147, 174, 203, 205, 208, 219, 220
pointer, *43*
as argument, 335
arithmetic, 45
for built-in array index, 420
incorrect, 419, 420
and array, 44
as array origin, 420
as array, example of use, 377
to automatic objects, 192
to base class, 236
to build aggregate, 420
built-in, 62
as data member, 163
compared to programmer-defined, 427
as parametric type, 330
as member, 149, 150
memory management problems, 421
problems with exceptions, 423
call to member function through, 89
to class object, 331
class, 165
design of, 425
compared to reference, 51, 79
comparison, 45
const, 46
to const object, 46
copied versus counted, table, 425
copied-object, 425
as cursor, 420
declaration style, 77
dereference, 511
to dynamic object, 97, 108, 193
essential for "is-usable-as" relation, 276
to interface base class, 246
member datum, example of, 410, 421
to member, *513*
to member operator, 31, *513*
example of use, 531
to member function, example of use, 529
multiple use, 420
null, 44, 427
returned by ANSI C string functions, 544
programmer-defined, 419, 424
schematic diagram, 43
table of operators, 49
type, 43
use errors, 191, 420
use of object through base class, 241
uses, summary, 419
as view of object, 333
Points, 437
Polivka, R. P., 7, 535
Polygon, 508
polymorphism, *4*, *327*; *See also* virtual—function.

polynomial evaluation, 64, 84
polynomial root, example of, 111
Porsching, T. A., 218
Positive, *484*
positive definite matrix, *452*
positive integers as semigroup, 475
Positive<int>, 482
Positive<T>, 483, 484
PositivePredicate, *484*
post cleanup, 176
postdecrement operator, 31
postfix operator, programmer-defined, 165
postincrement operator, 31, 41
pow(), 22, 23, 480, 482, 483, 488, 490
  and algebraic structure, 481
Pratt, T. W., 195, 218, 415
precedence, 30, 45, *71*
  programmer-defined operator, 100, 160
predecrement operator, 31
predefined type. *See* built-in—type.
Predicate, 484
prefix operator, programmer-defined, 165
preincrement operator, 31
preinitialization, 175
Premerlani, W., 10, 261
preprocessor, *120*
  command, *15*
Press, W. H., 36, 47, 60, 313, 416, 537, 541, 571, 587, 595, 604, 611, 613–615
printf(), 84
printing, 15, 68
private, 87, 89, 110, 168, 207, 208, 211, 219, 236, 282, 290–293, 458
  base class, *282*, 292, 294, 299
    example of use, 285, 308, 374, 380
    for implementation inheritance, 295
    multiple, example of, 390
    of template class, 309
  copy constructor, example of, 376
  data members, *89*, 230
  default constructor, 269
  member, 208. *See also* access control.
  member function, *98*
    role of, 291
private:. *See* access—specifier.
production, *527*
program, *113*
  the big picture, 113
  design, 7
  termination, and destructors, 182
programmer-defined operator
  precedence and associativity of, 100
programmer-defined control structure, *415*
programming languages, 3
programming object, 227
project(), 366, 367, 368
projection, 312, *366*
  allowed conversion to reference, 385
  array interfacer, 405
  array, function declarations, 375
  concrete array, 392
    object sketch, 395
  example of use, 368, 406
  of interfaced arrays, 405
  as lvalue, 367
  matrix, 566
  operator, 165
    example of, 266, 267
  as reference, 369, 379
  relation to array reference, 388
  repeated, for subscripting, 367
  return type, 367
    in array interface, 403
  subscriptor, packed array example, 561
  template parameter recursion and, 393
ProjectionT, 367, 368, 398, *561*
promotion
  for argument matching, 134
protected, 236, 290–293. *See also* access control.
  constructor, example of, 376
  member datum, example of, 376
  member function, constructor, 292
  uses for, 292
protected:. *See* access—specifier.
  example of use, 307
prototype. *See* function—prototype.
pseudoinverse, *570*, 604
  solution of linear least squares, 570
ptrdiff_t, 315
public, 87, 168, 207, 236, 266, 290–293
  base class, 244, *266*
  derivation
    of implementation (inheritance), 294
    type relationship, 334
  implementation base class, 299
  inheritance of implementation, 270
  member, 208
    as software connectors, 290
public:. *See* access—specifier.
pure side-effect, *124*
pure virtual function, *234*, 356
  called in constructor, 261
  defined in derived class, 236
  definition, 261
  example of, 234, 315

Q

QuadraticComplexRoots, 111
QuadraticPolynomial, 110, *111*
QuadraticPolynomial::RealRoots, 113
QuadraticRealRoots, 111
qualification. *See also* scope resolution operator.
    calling pure virtual with, 261
qualify, *112*
question mark, 25

R

\r, 25
Rall numbers, *591*
Rall, L. B., 589, 591, 591, 594, 594
Rall1d<T, V>, *591*, 592, 593, 614
    as public base, 593
RallInvalidUnitErr, 593
RallT, 599, 602
random deviate, 123
Rational, *183*, 189, *194*, 195–197
rational numbers
    example, 189, 194
RationalComplex, *184*, 195–197
RationalInitializer, *194*, 195
READ, 15, 62
read-only array. *See* array—const-like.
readability, 64
reading complex declaration, 47
reading input, 68
    with while loop, 35
REAL, 15, 21, 554
REAL*16, 22
REAL*4, 22, 554
REAL*8, 22, 554
RealRoots, *111*, 112, 113
recompilation, 260
RectLURep<T>, *466*, 557, 572, 581, 605
RectSVDRep<double>, 606, 607
RectSVDRep<T>, *572*, 573, 581, 604, 605
    as public base, 604
recursion, *59*, 65
    copy constructor, generated, 148, 269
    default construction, 269
    destructor, generated, 269
    template parameter, 373, 390
reduction
    of array elements by operators, 534
reference, *20*, *50*, *69*, *78*, *98*
    as argument, 335
    as association, 20
    to base class, 236, *270*
    as built-in parametric type, 330
    built-in, as model for class, 379
    compared to pointer, 51, 79
    to const object, 51, 79
    const, initialization, 332
    essential for "is-usable-as" relation, 276
    formal argument, 81, *127*
    formal argument, example of use, 166
    as function return type, 98, 131
    initialization required, 51, 79
    to interface base class, 246
    member datum, 148
        to base class, example of, 247
        example of, 214, 249, 254, 277
        initialization, 248
        sketch, 248
    to temporary object, 340
    type, 51, 79
    use of object through base class, 241
    variable, 84
    versus pointer, for calls through interface base class, 250
    as view of object, 333
reference count, *425*, 446
reference-counted object pointer class, *425*, *431*
ReferenceCount, 433, *434*, 435
referential aggregation, *420*
    controlling object lifetime, 421
    and interfaces, 441
    pointer class for, 431
    sharing object lifetime, 423
    of static objects, 437
Reid, J., 218, 415
relational operator, 23, 31
    table of, 24
releaseControl(), 470
Rep, 464, 465, 466
repeat(), 480, 483, 488
    and algebraic structure, 483
repeated composition
    example, 488
repeated multiplication. *See* pow().
repeatedComposition(), 488
representation, 227
    ambiguity, 229, 584
    mesh example, 200
Resnick, R., 494
resolves, *285*
resource acquisition is initialization, *195*
resource management
    using automatic objects for, 187
return, 14, 52, 60, 131
    by reference, 131
        danger of, 132

return *(cont.)*
by value, 148
value of main program, 14, 17
return *this
example of, 549
return type, *122*
from array projection, 367
built-in pointer, problems with, 548
and function overloading, 132
none for constructor, 145
none for destructor, 165
not allowed for conversion operator, 152
not used for template matching, 139
operator, 161
operator functions, 160
of overloaded functions, 123
pointer versus object, 544
qualified with ::, 368
as range of function, 122, 516
reference, 98, 131
for search functions, 159
templatized, 321
virtual function, 398
void, 131
reuse, 4, 5
revision
to add interface, 280
to avoid code replication, 323
to capture implementation commonality, 278
to capture is-usable-as commonality, 237
of classes, 227
of classes to add virtual functions, 234, 258
to factor out template parameter independent implementation, 307
to factor out template class member function, 308
to group private members into a new class, 291
to increase encapsulation, 281, 299
to introduce protected members, 292
of member function, to use abstraction, 245
for more dynamic data structures, 437
to put functions in interfaces, 291
to renest classes, 316
to replace replication by templates, 309
to support in-place matrix factorization, 459
to use interface reference datum, 246
Richards, P. G., 575, 575, 603, 604, 613
right singular vectors, *570*
right-to-left binary method, 488
rigid array, 312
RigidArithmetic1d<double, 6>, 591
RigidArithmetic1d<T, n0>, *503*, 504, 593
RigidArray1d<float,5>, 336, 337
RigidArray1d<float>, 358
RigidArray1d<T, n0>, 504
RigidArray2d<double, 5, 5>, 350
RigidArray2d<int>, 358
RigidArray2d<T, n0, n1>, *322*
RigidRall1d<T, nvars>, *593*
RigidRall2d, 614
ring, *475*, 476
ring with unit, *475*, 476
RingCategory<T>, *491*, 505
RingWithUnitCategory<T>, *491*, 505
Ritchie, D., 67
Robson, D., 6, 261, 415
robust code, 219
robust estimator, *615*
Roman, P., 505, 508
row(), 367
row-major, 370
array, 42
order, 313
subscripting, 364, *370*
Rumbaugh, J., 10, 261
runtime
binding, *327*
error, array size mismatch, 366
expressions, example, 518
memory allocation. *See* dynamic object.
polymorphism, 327
sizing of arrays, 47

S

Saks, D., 262
SAXPY, 136
scalar, *476*, 492
as algebraic structure, 475
multiplication, 492
scanf(), 84
schizophrenic behavior, 275
deliberately chosen, 270
SciEngErr, 257, *258*, 317
as public base, 258, 307, 316, 529, 541, 555, 585
scope, 194, 285, *286*
and argument matching, 133
block, 194
class, 112, 178
for enumeration, 173
and derivation, 286
file, 178
and linkage, 179
friend declaration, 168
and function overloading, 132
global, 194

going out of, 98
injecting names into, 499
and linkage, 178
local, 56, 178, 180, 194
nested class, 112
and object lifetime, 178
operator. *See* scope resolution operator.
and order of construction, 181
qualifier. *See* scope resolution operator.
static member, 94, 95
scope resolution operator, 31, *91*, 94, 110, 112, 286
in access declaration, 293
in actual template parameter, example of, 391
to call static members, 167
example of, 94, 167, 267, 269, 288, 464
for global function, 549
example of use, 531
for member function definitions, 166
for member function pointer, 513
on members of nested classes, 113
parameterized, example of use, 310
for physical constants, 495
return type, example of, 112, 321, 368
template parameter and, 321
on template parameter type, example of, 599
searching intervals, example, 157
secret, *207*, 215, 219, 274, 282
representation, 230
security versus encapsulation, 219
semicolon, 13, 20, 86
semigroup, *474*
algebraic category, 485
SemiGroupCategory<ComplexFloat>, 486
SemiGroupCategory<T>, *485*, 486, 487
as public base, 488, 491
SemiGroupCommon<T, Structure>, 487
separate compilation, 114, 119
separation of object design and use, 230
separation of interface and implementation, 260, 294, 300
to avoid errors, 276
opposite of extension by public inheritance, 270
sequence point, *578*
set/get functions. *See* get/set functions.
setbuf(), 83
setiosflags, 29, 74
SetOfIntervals, *157*
setprecision(), 29, 74
setToOne(), 479
setToZero(), 479
setw, 29, 35, 74
setw(), 552, 553
SGETRF, 456, 460
shallow copy, *149*
shape, array, *312*, *316*
shared
aggregation, *421*
data, 93
object via member datum reference, 251
shift, left or right, 27
short int, 22, 23, 134, 554
short-circuit evaluation, 24
SI (*Système International*) units, 495
SIConstants<double>, 499
SIConstants<T>, 495, *498*
as private base, 494
side effect, *124*, 125, 139
avoiding
with call by value, 125
with const reference arguments, 128
classes and pure, 125
input modification, 124, 125
problems with, 125
pure, 124
reference argument, 127
silver bullet, 10
SimpleArrayId<int>, 333
SimpleArrayId<T>, 136, 328
SimpleArray2d<T>, 173
SimpleArray<double>, 306, 322
SimpleArray<Element>, 203
SimpleArray<float>, 305, 310, 346–349
SimpleArray<int>, 197, 305, 306, 310, 332, 333, 346–349
SimpleArray<Point>, 203
SimpleArray<Subscript>, 328
SimpleArray<T>, *101*, 102, 107, 108, 117, 128, 136, 159, 173, 203, 266, 303–306, 310, 313, 328, 346, 349
as public base, 309
SimpleArrayShape, 328
SimpleDoubleArray, 100
SimpleFloatArray, 95, *97*–102, 117, *149*, 150, 163, 266–270, 301
as public base, 266
SimpleIntArray, 100
SimpleTrig, *514*
simulator
exercise, 263
example, 250
SimulatorFactory, *253*–255, 262
as public base, 255
simultaneous equations
damped least squares, 603
nonlinear, 594

simultaneous equations *(cont.)*
representation for, 598
single derivation, *300*
name hiding is dominance for, 288
single quote, 24, 25
singular matrix, inverse of, 571
singular value decomposition, *569*; *See also* SVD.
example of use, 609
and nonlinear least squares, 603
singular values, *570*
size, *312*
of data variable at runtime, 175
sizeof, 31, 44, 70, 160
size_t, 315
Sloane, A. D., 139
small program versus large program, 205
Smalltalk, 195, 327
compared to C++, 6
smart pointer, *424*, 446
Smith, B. T., 451
Snyder, A., 300, 358
software engineering, *10*
Sommerville, I., 10
Sorensen, D., 370, 451, 452, 467, 559, 579
source code, 9
file names, 68
FTP address for, 9
index. *See* separate Source File Index
layout, 13
special member function. *See* assignment, constructor—copy, constructor—default, copy constructor, copy assignment, destructor.
specialization, function template, 139
sprintf(), 75
SPSTRF, 456
sqr(), 22, 55, 82, 104, 105
sqrt(), 17
example of use, 35, 92
square-brackets operator. *See* operator[ ], projection—operator.
for string class, 543
Srinivasan, V., 200
sscanf(), 75
stack, 185
standard
conversion, *134*, 138
and argument matching, 134
error, 27, 72
header file
ctype.h, 25
float.h, 22, 574
fstream.h, 30
iomanip.h, 29
iostream.h, 15
limits.h, 22
math.h, 17
stddef.h, 76, 83, 315
stdio.h, 68
stdlib.h, 60, 543
string.h, 106, 543
sys.h, 114
I/O library, 542
input, 15, 27, 68, 72
math library, *17*
output, 15, 27, 68, 72
start symbol, *527*
state, 4, *74*, *229*, 268
ambiguous representation of, 230
for design not use, 230
in implementation base, 271
related to behavior, 229
statement, *13*
declaration, 38
static, 92, 123, 124, 132, 166, 167, *176*–180, 184, 185, 484, 563
constructors. *See* construction, order of.
member datum, *94*
example of, 94, 177
member function, 95, *166*
example of, 485, 557
object
construction, 181, 183
destruction, 181
initialization, 195
lifetime, 176, 178
and linkage, 178
local scope, initialization of, 180
local, construction order, 182
local, destruction order, 183
nonlocal, construction order, 182
nonlocal, destruction order, 182
private, 194
stddef.h, 76, 83, 315
stdio.h, 68, 72, 74, 542
and C++ I/O, 542
stdlib.h, 60, 105, 543
Steele, Jr., G. L., 25, 62, 578, 579
steepest-descent, *604*
stepwise refinement, 10
Stewart, G. W., 136, 451
Stirling number, 59
stochastic inverse, *613*
STOP, 555
Straker, D., 61, 84
strcat(), 544, 579

strchr(), 544
strcmp(), 106, 256, 544, 551
strcpy(), 544, 549
strcspn(), 544
stream, *27*, *71*, *535*
stream state, 172
stride, array, *370*
  in an assignment, example of, 395
String, 258, 484, 529, 539, 543, *545*–553, 578, 579
  concatenation, 542, 543
  design of, 543
  example of, 68, 542
  initialization, 50
  as monoid, 475
  searching functions, 546, 551
string.h, 106, 178, 256, 543
strlen(), 544
strncpy(), 178, 544
Stroustrup, B., 5, 7–11, 62, 77, 115, 139, 190, 195, 261, 262, 300, 356–358, 505, 527, 578, 628, 650, 651
strpbrk(), 544
strrchr(), 544
strspn(), 544
strstr(), 544
strtod(), 544, 552
strtok(), 579
strtol(), 544
strtoul(), 544
struct, 115
structural commonality, 104
structured
  grid, *219*
  mesh, *219*
  programming, 10
style
  avoid unions, 21
  break and continue statements, 37
  conditional operator, 33
  consistent, 61
  declare derived class virtual functions, 237
  if statement, 32
  indentation, 61, 83
  local variables, 57
  one statement per line, exception, 366
  one variable per declaration, 77
  parentheses and spacing, 30
  pointer declarations, 62
  variable declarations, 39, 75
subclass, *358*
SubDomain<Domain, Predicate>, 483, *484*
  as public base, 484
subobject
  base of derived class, 266
SUBROUTINE, 52
  as abstraction, 206
subroutine library, 3, 9, 15, 136, 539
subscript, 40, 322, 372, 393
Subscript, *315*
SubscriptArray<1>, 348
SubscriptArray<2>, 348, 379, 569
SubscriptArray<ndim>, *347*, 348, 372
  example of use, 372
subscripting, array
  issues in design of, 318
  operator, 31; *See also* function call operator, projection—operator.
  example of, 322
  random access versus sequential, 407
  by repeated projection, 367
  signed versus unsigned, 315
Subscriptor, 369–371, *373*–377, 391–393, 396, 415
  array, 373
  connection between projection and array, 391
  object, returned by projectionSubscriptor(), 393
  packed array example, 560
  as private base, 374, 380, 382
  projection, 389
  template parameter, 370
  transposed, 568
subscriptor(), 415
SubscriptRange, *316*
SubscriptRangeError, *108*, 110, 258, 266, 309
substitutability of types principle, 357
subtype, 358; *See* interface base class.
  inclusion polymorphism, *327*
summing array elements
  example, 129
sumsq(), 336
Sundaresan, C. J., 539, 577
+(superscript on matrix). *See* pseudoinverse.
Sussman, G. J., 535
Sutor, R. S., 261, 327, 357
SVD (Singular value decomposition), *569*
  example, 571
  fitting polynomials example, 576
  linear least squares, example, 575
swapping function arguments, 59, 81
  example, 127
Symmetry, *316*
symmetry group, 507
symmetry operations, 507
SymPosDefPackedLURep<T>, *467*, 560
SymPosDefPackedLURep<T>::Factored, 579
synchronizing
  I/O from C and C++, 542

Synder, Alan, 300
synonym
  typedef, 87
  for type, 40
syntax, 8, 62
SyntaxErr, *529*
system of classes
  advantages, 363
  array, 363
  benefits, 563
Système International, *495*, 498

T

\t, 25
Tang, J.-M., 200
Taylor series
  for Newton-Raphson algorithm, 594
  for nonlinear least squares, 587
Taylor, R. H., 508
TaylorCoefficient1d<T>, 588, 599
Teale, S., 29, 30, 74, 75, 577, 579
technical programming. *Short for* scientific and engineering programming.
template, 9, 101–104, 115, 136, 304
  argument, *101*, 104, 136
    class, 101
    compile-time constants, 101
    int, example of, 117
    template type, 117
    type, 101
  and base class composition, 323
  category, 314, *326*
  class, *101*, *304*
    as derived class, 307
    8 parameters, example of, 497
    name, *346*
    nontype parameters, example of, 497
    parameter recursion, example of, 373
    type, 348
    versus class template, 101, 304
  constant expressions as arguments, 323
  definition. *See* class template, function template.
  essential, 116
  expansion, 321, 326
    avoiding with inline functions, 324
    base class member function, 310
    base class templates and, 310
    controlling, 354
    injecting friend declarations into global scope, 354
    reducing, 309
    restricted, 351
    type checking, 348
  function, 136, *304*
    example of, 383, 468, 533
    versus function template, 304
  implementation base class, 323
  interface base class, 313, *320*
    example of use, 322
  interface category, *320*
  as macro, 303
  member function independent of parameter, 307
  nontype parameter, 322
  parameter. *See also* template—argument.
    as member datum type, example of, 401, 592
    as member datum, example of, 523
  programs without, 327
  recursion, 390
  restrictions on parameters, 304
  specialization, *139*, *348*
    avoided, 557
    class, example of, 347, 373
    type relationship, 349
  versus derivation, 326
  versus interface base class, 326
temporary object, 152, 165, *340*
  accessed array projection, 406
  array reference, example of, 393
  assignment to, example of, 395
  in binary operator, 486
  constructed in return, example of, 393
  conversion operator, 339
  created by constructor call, 379
  elimination of, 505
  explicit, to prevent conversion, example of, 593
  pointer object problem, 447
  problems with, 548
  requires const reference, 340
  and sequence point, 578
termination
  by throw, 106, 107
terminology, 10
Teukolsky, S. A., 36, 47, 60, 313, 416, 537, 541, 571, 587, 595, 604, 611, 613–615
this, 92
  example of use, 216
*this
  object reference, example of use, 386
  return *this, example of, 269
Thompson, D. W., 206
throw, 105, 106, 185, 257

example of use, 106, 159, 267, 273, 541
specification, *140*
tight coupling, and friend classes, 171
tilde, 96
TokenKind, 528
tokens, *527*
TooLittleDataInput, *585*
top-down, *10*
Touzot, G., 218
TraceLifetime, *177*, 178, 181
as public base, 447
trade-off, available in system of classes, 363
traditional C, 67
transformation of collection elements, 534
translation unit, *119*, 123
consistency across, 120
and header files, 120
initialization of objects in, 194
and linkage, 178
object construction order in, 182
objects shared between, 179
order of static constructors across, 196
transpose
array, relation of FORTRAN to C++, 553
matrix, example, 568, 573
TransposedConcreteBlas2d<T>, *568*, 573
Treitel, S., 603
trivial conversion, *133*
.TRUE., 24
try block, 107, 109, 110
example of use, 107, 112, 259, 556, 592
type, *18*, 21, 38, 69, 329, 330
base, versus subtype, 358
for both object and variable, 332
built-in, 70, 330
cast, *342*
operator, 31
checking, 20, *21*, 40, 53, 54, 114, 119, 335
const references, 128
role of header files, 120
at runtime, 464
templates, after expansion, 348
and throw specification, 140
value, 329
virtual functions, 236
class, 85, 330
code, for type testing, 341
const, 331
conversion, 40, 54
automatic, 54
of function arguments, 126
in function call, 126
rules, *21*
creation, 335
determining function call, 275
encapsulation and, 357, 358
family of closely related, 100
information
amount needed, 368
loss, 340
recovery, 343
object versus reference to, 334
parameterized, 101
and physical quantities with units, 496
programmer-defined, 85
relationship, 332
base class reference to derived-class object, 338
const reference, 333
conversion operator, 339
derived class and base class template, 349
indirect view versus value conversion, 333
and indirection, 333
public derivation, 334
template specialization, 349
templates, 346
templatized interface, 349
usual arithmetic conversions, 330
restriction on function template, 352
as rules, 329
runtime modification, 460
as set of objects, 357
synonym for, 40
template parameter, 350
testing
example of, 342
problems, 343
theory of, 357
use, 335
typedef, *40*, 87, 315, 321, 322, 331, 364, 466, 505, 511–513
for abbreviation, 331, 364
example of, 607
function pointer, example of, 598
for function type, 511
for member function pointer, 513
example of, 529
for name commonality, 518
nested in class template, example of, 321
nested in class, example of, 373, 375
as symbolic name, 466
in template as common name, 320
with template for robust code, 512

U

unary operator, 31
UnaryFunctional<Domain>, 523
unformatted I/O, 62
Ungar, D., 195
unhide, *288*, 297
  example of, 288
union, 20, 21
  danger of, 21
unit, *475*
units (dimensional analysis)
  derived from fundamental, representation of, 498
  fundamental, representation of, 498
  fundamental, SI system, 495
  MKS system, 498
  on physical quantities, 493
Unknowns1d, 466
Unknowns2d, 466
unprintable character, 24
unsigned int member datum, example of, 434
updating assignment operator, *26*
uppercase, 19
use type, *335*
use versus design, 230
UsedInInvalidStateErr, *158*, *307*
user-defined conversion, *135*, 151, 152
  and argument matching, 135
user-must-define functions, 354, 482
  example of, 497, 519, 591
UsesSA<ndim>, *348*
usual arithmetic conversions, *23*, 26, 54, 126
  as type relationship, 330

V

\v, 25
value, *18*, *68*, 69
Van Loan, C. F., 417, 456, 470, 571
Vandermonde, 417
Vandermonde matrix, *417*
variable, *18*, 68, *69*
  assignment, 58
  associated with object, 38, 57
  creation, 38
  declaration, 19
  initial value, 20
  name, 13
  schematic diagram, 69
  scope, 57
  type, 332
vector, *476*, 492
  as algebraic structure, 475
vector space, *476*
Vector<T>, 136
vertical tab, 25
Vetterling, W. T., 36, 47, 60, 313, 416, 537, 541, 571, 587, 595, 604, 613–615
virtual, *231*, 234–237, 257, 260, 273, 275, 281, 283, 299, 400, 447
  base class, *281*
    avoiding with code replication, 491
    in base class composition, 294
    criterion, 301
    and diamond-shaped class DAG, 296
    downcast prohibited from, 343
    efficiency, 300
    example of, 280, 281, 285, 293, 318–324, 342, 344, 398, 400, 401, 404, 405, 442, 461, 464, 516–518, 521–525, 585–599, 607, 610
    to extend interface, 295
    how it works, 282
    initialization of, 300
    with member data, 300
    member function calls, 283
  constructor, *442*
  destructor, 234, 257, 443
  factory. *See* object—factory.
  function, 260, 287
    call through reference, example of, 249
    called by base class member, 273
    clone. *See* clone function
    definition, 237
    in derived class, example of, 236
    in derived class, example of, 236
    design decision for base class, 275
    effect of, example, 235
    efficiency of, 299
    example of use, 234, 273
    hidden, 287
    impact on clients, 275
    inline, 299
    match of derived and base class, 237, 262
    not in interface, example of, 572, 604
    overloading, 287
    overriding, example of, 272
    return type, 398, 447
    unhiding, 288
    when not to use, 275
visualization of objects, 238
Vlissides, J. M., 262
void, 52, 53, 131
  as argument type, 131
  as return type, 131
void*, 83, 151

voltage supply object, 231
VoltageSupply, 231, 233, *234*–250, 263, 271, 275–279, 282, 286–291, 295, 296, 301, 318, 334, 335, 340
  as public base, 235, 236, 244, 247, 278, 294
  as public virtual base, 281, 285, 293, 342, 344
Voltmeter, 245, 248–250, 263
voltmeter object, 231
VoltOn59, 231, *232*–236, 245, 263, 279, 301, 334
VoltOn59Dual, 295, 296
VoltOn59_VS, 235, *236*–243, 334, 335, 340
VoltOn59_VS_GI_GC, 263, 277, 278
VoltyMetrics, *232*, 241, 245, 251, 263, 279, 301
VoltyMetricsSimulation, 251, *252*, 253, 257, 263
VoltyMetrics_VM_GI, 256
VoltyMetrics_VM_GI_GC, 263, 277, 278

W

warning
  arrays start at 0, 41
  avoid single argument constructors, 156
  a[i,j] is not 2D, 165
  built-in pointer member data shares objects, 150, 164
  check for self assignment, 164
  equality test versus assignment, 23
  floating point constants, 23
  integer constants, 21
  member initialization order, 147
  minimize conversions, 154, 156
  m[1,2] is not 2D array access, 42
  pass constant built-in arrays as const arguments, 130
  return by reference, 132
  self assignment, 99
Wegner, P., 262, 327, 357
Weiss, D. M., 207
while-loop, 34, 37
  example of use, 45, 98, 150
whitespace, *28*, 72
Wiener filtering via stochastic inverse, 613
Wiener, Lauren, 10
Wilkerson, B., 10, 208
Wing, J. M., 357
Wirfs-Brock, R., 10, 208
Wirth, N., 10
Wolfram, S., 7, 535
wrapping
  BLAS subroutines, example, 563
  legacy code in objects, 577
  non-C++ function, 539
WRITE, 15, 62, 63

X

X, *168*
xdot (BLAS), example of use, 567
XERBLA, 555
XFunctional<Domain>, 521, 523
XFunctional<double>, 531
xgemm (BLAS), example of use, 566
xgemv (BLAS), example of use, 565
xgesvd (LAPACK), example of use, 573
xscal (BLAS), example of use, 565

Y

Y, *168*
YazooMatrix, 331
Yourdon, E. N., 10, 619, 620, 624

Z

Z, *168*
ZAXPY, 136
Zdonik, S. B., 357
Zelkowitz, M. V., 617
zero
  is false, 24
  as null pointer, 44
Zienkiewicz, O. C., 218

SOURCE FILE INDEX

A

Algebra/AbelianGroupCategory.h, 490
Algebra/AbelianSemiGroupCategory.h, 486, 487
Algebra/ComplexFloat.h, 479, 486
Algebra/DivisionAlgebraCategory.h, 493
Algebra/ExternalScalarsCategory.h, 492
Algebra/FieldCategory.h, 491
Algebra/FieldScalarsCategory.h, 493
Algebra/GroupCategory.h, 489
Algebra/GroupCommon.c, 490
Algebra/LeftScalarsCategory.h, 493
Algebra/MonoidCategory.h, 488
Algebra/MonoidCommon.c, 488
Algebra/Positive.h, 484
Algebra/RingCategory.h, 491
Algebra/RingWithUnitCategory.h, 491
Algebra/SemiGroupCategory.h, 485, 487
Algebra/SemiGroupCommon.c, 487
Algebra/SubDomain.h, 484
Array/AccessedArray2d.h, 404
Array/Array2d.c, 401
Array/Array2d.h, 398, 400
Array/ArrayIterator2d.h, 412
Array/ArrayShape.C, 316
Array/ArrayShape.h, 315
Array/ConcreteArray1d.c, 384, 395
Array/ConcreteArray2d.c, 377, 383, 384, 393
Array/ConcreteArray2d.h, 374–377, 380, 382, 384–386, 393
Array/ConcreteArrayIterator.h, 410, 411
Array/ConcreteArrayProjection1d.h, 391, 395
Array/ConcreteArrayShape.c, 394
Array/ConcreteArrayShape.h, 372–374, 389–391
Array/ConcreteFortranArray2d.c, 378, 379
Array/ConcreteFortranArray2d.h, 378
Array/ConcreteFortranSymmetricPacked-Array2d.C, 562
Array/ConcreteFortranSymmetricPacked-Array2d.h, 560–562
Array/FormedArray2d.h, 402
Array/InterfacedArray2d.h, 401, 405, 406
Array/SubscriptArray.h, 347
AutoDeriv/Rall1d.c, 592, 593, 599
AutoDeriv/Rall1d.h, 591, 592
AutoDeriv/RigidRall1d.h, 593
AutoDeriv/TaylorCoefficient1d.h, 588

Ch2

ch2/array-reverse.C, 45
ch2/array-zero.C, 45
ch2/bitwise.C, 26
ch2/break.C, 37
ch2/charArray.C, 50
ch2/charcomp.C, 25
ch2/commandLine.C, 60
ch2/comments.C, 13, 14
ch2/const.C, 39
ch2/cosang.C, 17
ch2/coulombsLaw-onefile.C, 52
ch2/coulombsLaw.C, 54
ch2/coulombsLaw.h, 53
ch2/coulombUse.C, 53, 54
ch2/dotproto.C, 64
ch2/doubleSqr.C, 55
ch2/enum.C, 26, 63
ch2/ex-ptrs.C, 63, 64
ch2/ex-refs.C, 64
ch2/ex-varsketch.C, 63
ch2/ex-while.C, 63
ch2/fortran-compatible-io.C, 29
ch2/funcarg.C, 57–59
ch2/ifthel.f, 16
ch2/includes.C, 54
ch2/intercepts.C, 16, 17
ch2/io.C, 29
ch2/iodemo.C, 27, 28
ch2/linefit.C, 48, 49
ch2/mat3by3mult.C, 41
ch2/mat3by3mult2.C, 57
ch2/Newton.C, 36
ch2/prepostfix.C, 22
ch2/ptrArray.C, 44, 47
ch2/ptrconst.C, 46
ch2/ptrdclstyles.C, 62
ch2/ptrs.C, 43, 44
ch2/refs.C, 50, 51
ch2/regurgitate.C, 15
ch2/scope.C, 56
ch2/simple.f, 18, 19, 39, 40, 47
ch2/simplecpp.C, 19, 20, 32, 33, 37, 38, 41, 42
ch2/sqrtTable.C, 35
ch2/Stirling.C, 60
ch2/tempct.f, 32

ch2/tempctrl.C, 32, 33
ch2/trivial.C, 14
ch2/typedef.C, 40
ch2/vertstyle.C, 61

Ch3

ch3/char-out.C, 70
ch3/const.C, 76
ch3/define.c, 75
ch3/doubleSqr.C, 82
ch3/enum.C, 70, 71
ch3/ex-refs.C, 84
ch3/fortran-compatible-io.C, 73
ch3/funcarg.C, 80, 81
ch3/heap.c, 77, 78
ch3/input.c, 72
ch3/io.C, 75
ch3/iodemo.C, 72, 73
ch3/new.C, 77, 78
ch3/ptrdclstyles.C, 77
ch3/refs.C, 78, 79
ch3/regurgitate.C, 67
ch3/simplecpp.C, 69
ch3/sqrtTable.C, 74
ch3/vertstyle.C, 83
ch3/void.C, 80

Ch4

ch4/CheckedSimpleArray.c, 108
ch4/CheckedSimpleArray.h, 108
ch4/cLine.C, 115
ch4/distance.C, 87
ch4/distance2.C, 116
ch4/FixedArray.C, 117
ch4/interactive.C, 106
ch4/interactive2.C, 106, 107
ch4/intersect.C, 90, 94
ch4/Line.C, 93, 94
ch4/Line.h, 90
ch4/linefit.C, 95
ch4/linefit2.C, 102
ch4/linest.f, 115
ch4/Nested.C, 110
ch4/Point.C, 91, 92
ch4/Point.h, 87
ch4/Point0.C, 86
ch4/ptrselect.C, 89
ch4/quadratic.C, 111–113
ch4/SimpleArray.c, 102
ch4/SimpleArray.h, 101
ch4/SimpleFloatArray.C, 97–99
ch4/SimpleFloatArray.h, 97
ch4/sqr.C, 105
ch4/tCheckedSimpleArray.C, 109
ch4/tPoint.C, 91

Ch5

ch5/abs.C, 133
ch5/args.C, 125, 126, 128, 131, 132, 134, 135
ch5/arrayArg.C, 129, 130
ch5/axpy.c, 137
ch5/axpy.h, 136
ch5/blastemp.C, 137
ch5/con-callbr.C, 128, 129
ch5/dcldef.C, 119–124
ch5/dot.C, 139
ch5/expdev.C, 123
ch5/hasVal.C, 140
ch5/logof.C, 130
ch5/resolve.C, 141
ch5/retdcl.C, 131
ch5/swap.C, 127, 140
ch5/taxpy.C, 138
ch5/taxpy1.C, 138
ch5/wrong1.C, 141
ch5/wrong2.C, 141

Ch6

ch6/ambig-convert.C, 153
ch6/ambig-convert2.C, 154, 155
ch6/ambig-convert3.C, 156
ch6/arrayAssign.C, 155
ch6/Bool-examples.C, 151, 152
ch6/Circle.C, 146, 147, 149
ch6/Circle.h, 145
ch6/cmplx.C, 161, 162
ch6/cmplx1.C, 162
ch6/cmplx2.C, 160, 161
ch6/CmplxFlt.C, 166
ch6/dcl-syntax.C, 146
ch6/demoFal.C, 157–159, 171, 172
ch6/demoList.C, 169
ch6/friend.C, 167, 168
ch6/List.h, 170
ch6/mult-assign.C, 164, 173
ch6/Point.h, 144
ch6/SimpleFloatArray.C, 150, 163
ch6/SimpleFloatArray.h, 149
ch6/UsePoint.C, 144

Ch7

ch7/cache.C, 181

ch7/class-scope.C, 179, 180
ch7/class-scope.h, 179, 180
ch7/Dangle.C, 191–193
ch7/mixedNesting.C, 196
ch7/newDemo.C, 189, 190
ch7/Rational.C, 184
ch7/Rational.h, 183
ch7/Rational1.C, 195
ch7/Rational1.h, 194
ch7/RationalComplex.C, 184
ch7/RationalComplex.h, 184
ch7/resource-acq.C, 187
ch7/static-linkage.C, 179
ch7/tAutoException.C, 186
ch7/tAutoNesting.C, 185
ch7/tDynLifetime.C, 197
ch7/TraceLifetime.C, 177
ch7/TraceLifetime.h, 177
ch7/tStaticNesting.C, 181
ch7/zero.C, 184

Ch8

ch8/solution1.C, 203–205
ch8/solution2.C, 208–211
ch8/solution3.C, 213–217

Ch9

ch9/Acme130Simulation.C, 253
ch9/Acme130Simulation.h, 251
ch9/Acme130_VS.h, 235
ch9/Acme130_VS_GI.C, 244, 245
ch9/Acme130_VS_GI.h, 244
ch9/Acme130_VS_GI_GC.C, 248
ch9/Acme130_VS_GI_GC.h, 247
ch9/checkCalibration_VS.h, 235
ch9/CheckedSimpleArray.h, 258
ch9/ExperimentSimulation.h, 252
ch9/GPIBController.h, 246
ch9/GPIBController_GC.h, 247
ch9/GPIBController_GIS.C, 254, 255, 257
ch9/GPIBController_GIS.h, 254
ch9/GPIBController_Stub.h, 226
ch9/GPIBInstrument.h, 244
ch9/GPIBInstrumentSimulation.h, 251
ch9/IVTester.C, 249
ch9/IVTester.h, 249
ch9/new-virt-ret.C, 262
ch9/SeeCommonality.C, 227, 228, 232, 233
ch9/SeeVoltageSupply.C, 235, 240
ch9/SimulatorFactory.h, 253
ch9/tGPIBInstrument.C, 243
ch9/tIVTester.C, 249
ch9/tSimulator.C, 255, 256, 259
ch9/VoltageSupply.C, 234
ch9/VoltageSupply.h, 234
ch9/VoltOn59_VS.h, 236
ch9/VoltyMetrics.h, 232
ch9/VoltyMetricsSimulation.C, 253
ch9/VoltyMetricsSimulation.h, 252

Ch10

ch10/Acme130_Acs.h, 293
ch10/Acme130_Fwd.h, 278
ch10/Acme130_GIData.C, 284
ch10/Acme130_GIData.h, 281
ch10/Acme130_VS_GI.C, 277
ch10/Acme130_VS_GI_GC.C, 274
ch10/Acme140.h, 271, 272
ch10/Acme230.C, 286, 288
ch10/Acme230.h, 285
ch10/CheckedFloatArray.h, 266–269
ch10/GPIBInstrumentData.C, 278
ch10/GPIBInstrumentData.h, 277
ch10/GPIBInstrumentData_GI.h, 280
ch10/tAcme130_Acs.C, 293
ch10/tAcme130_GIData.C, 292
ch10/tAcme140.C, 273, 274, 276
ch10/tAcme230.C, 286
ch10/tCheckedFloatArray.C, 268, 270
ch10/tNoV.C, 294
ch10/VoltOn59_VS_GI.C, 277
ch10/VoltyMetrics_VM_GI.C, 277

Ch11

ch11/Array2d-with-dim.C, 323, 324
ch11/ArrayErr.C, 317
ch11/ArraynT.C, 318
ch11/ArrayxD.C, 319
ch11/ArrayxD.h, 319
ch11/average.C, 320–322, 325
ch11/CheckedSimpleArray.h, 309, 310
ch11/rigid.C, 322, 323
ch11/sizet.C, 315
ch11/sum.h, 320
ch11/tTuplize.C, 305
ch11/tuplize.c, 305

Ch12

ch12/Acme130_GC.h, 344
ch12/Acme130_GCTC.h, 342
ch12/conversion.C, 339, 340
ch12/GPIBComponent.h, 344

ch12/GPIBComponent_TC.h, 341
ch12/not-equal.C, 351
ch12/pure-virtual-called.C, 357
ch12/sumsq.C, 336
ch12/tCheckedSimpleArray.C, 350
ch12/tEquality.C, 353, 354
ch12/tGPIBComponent.C, 341, 345
ch12/tGPIBComponentTC.C, 342
ch12/tMatrix.C, 331, 333
ch12/tMixApplesOranges.C, 359
ch12/tObjtype.C, 351
ch12/tRefsNObjs.C, 334
ch12/tSubscriptArray.C, 348
ch12/tSum.C, 349, 350
ch12/tTypeErrsEg.C, 331, 332, 357

Ch13

ch13/bad-assign.C, 400
ch13/do.f, 407
ch13/IteratorFrobenius.C, 408
ch13/tCollect.C, 380, 385, 388, 392, 406, 411, 412
ch13/tConcreteMax.C, 365–368
ch13/tDiagonal.C, 416
ch13/while.C, 408

Ch14

ch14/exceptionLeak.C, 423
ch14/exceptionNoLeak.C, 431
ch14/ptrDemo.h, 447
ch14/tCounted.C, 437–441
ch14/tMolecule.C, 442–445

Ch15

ch15/Driver.C, 467, 468
ch15/tmpl0.C, 457, 458
ch15/tmpl1.C, 460–463

Ch16

ch16/AtwoodsMachine.C, 494–496
ch16/global-category-template.C, 485
ch16/global-function-template.C, 485
ch16/GroupCategory.h, 489
ch16/lin-frac.C, 480
ch16/tASGArray.C, 500, 501
ch16/tGroupD3.C, 507
ch16/tIntMod5.C, 506
ch16/tSpaceVector.C, 506

Ch17

ch17/FAMaker.C, 530, 531, 532
ch17/FAMaker.h, 529
ch17/Lexer.h, 527
ch17/ptr-member.C, 513, 514
ch17/tCachingFunction.C, 536
ch17/tFAMaker.C, 532
ch17/tFunction.C, 518
ch17/tLegendre.C, 537
ch17/tNormalDensityFunction.C, 515, 516
ch17/tPtrSin.C, 511–513
ch17/tReduce.C, 534
ch17/tSharedMean.C, 526
ch17/tSimpleFunction.C, 520
ch17/tSimpleTrig.C, 514

Ch18

ch18/blank-concat.C, 543
ch18/demoString.C, 543
ch18/miscdcls.C, 540, 541, 553
ch18/nrerror.C, 541, 542
ch18/syncIO.C, 542
ch18/tempdel.C, 548
ch18/tLegendreFit.C, 576
ch18/tRectSVDRep.C, 575
ch18/tXERBLA.C, 556

Ch19

ch19/DualGaussianModel.h, 610
ch19/tDampedSVD.C, 611
ch19/tNewtonRaphson.C, 601, 602
ch19/tRall.C, 594

D

DataModeling/DampedSVDIteratedEquations.c, 608, 609
DataModeling/DampedSVDIteratedEquations.h, 607
DataModeling/DataModel.h, 589
DataModeling/DualGaussianModel.c, 610
DataModeling/FormedPhysicalDáta.C, 586
DataModeling/FormedPhysicalData.h, 585, 586
DataModeling/IteratedEquations.h, 595
DataModeling/LeastSquares.c, 606
DataModeling/LeastSquares.h, 605
DataModeling/LinearizationIterator.c, 597, 598
DataModeling/LinearizationIterator.h, 596
DataModeling/NewtonRaphsonIteratedEquations.c, 600

DataModeling/NewtonRaphsonIteratedEquations.h, 598, 599
DataModeling/PhysicalData.h, 584, 585

F

Function/BinaryFunctional.h, 522
Function/ConstantFunctional.h, 523
Function/Function.h, 517
Function/Functional.h, 516
Function/FunctionalAlgebra.c, 520, 522, 525
Function/FunctionalAlgebra.h, 518, 519
Function/FunctionApplication.c, 533
Function/GaussianFunctional.h, 525
Function/NormalDensityFunction.c, 515
Function/NormalDensityFunction.h, 515
Function/ParameterFunctional.h, 524
Function/reduce.h, 534
Function/tApply.C, 534
Function/XFunctional.h, 521

L

LapackWrap/BlasSubroutines.C, 555, 556
LapackWrap/BlasSubroutines.h, 166, 555, 563
LapackWrap/ConcreteBlas2d.c, 565–567
LapackWrap/ConcreteBlas2d.h, 564, 566
LapackWrap/DampedRectSVDRep.c, 605
LapackWrap/DampedRectSVDRep.h, 604
LapackWrap/Lapack.h, 464, 465
LapackWrap/LapackSubroutines.h, 554, 557, 558
LapackWrap/RectLURep.c, 559
LapackWrap/RectLURep.h, 466
LapackWrap/RectSVDRep.c, 573, 574
LapackWrap/RectSVDRep.h, 572
LapackWrap/SymPosDefPackedLURep.h, 467
LapackWrap/TransposedConcreteBlas2d.c, 568, 569
LapackWrap/TransposedConcreteBlas2d.h, 568

S

SciEng/ArrayErr.h, 316
SciEng/Boolean.h, 151
SciEng/CloneableObjPtr.c, 444
SciEng/CloneableObjPtr.h, 443
SciEng/CopiedObjPtr.c, 427, 429
SciEng/CopiedObjPtr.h, 426, 430
SciEng/CountedObjPtr.c, 435, 436
SciEng/CountedObjPtr.h, 433, 436
SciEng/EquivalentCategory.h, 352
SciEng/Fallible.C, 309
SciEng/Fallible.h, 307, 308
SciEng/NumericalLimits.h, 574, 575
SciEng/ReferenceCount.h, 434, 435
SciEng/SciEngErr.C, 258
SciEng/SciEngErr.h, 258
SciEng/String.C, 549–552
SciEng/String.h, 545–547, 579
SciEng/Subscript.h, 315
SciEng/tString.C, 544
SciEng/utils.h, 104

U

Units/DerivedUnits.h, 498
Units/FundamentalUnits.h, 498
Units/Physical.h, 497, 499
Units/SIConstants.c, 499
Units/SIConstants.h, 498

V

Vector/Distribute.h, 502
Vector/DistributingAbelianSemiGroup.h, 501
Vector/DistributingDivisionAlgebra.h, 503
Vector/RigidArithmetic.h, 503

9 780201 533934 55595

ISBN 0-201-53393-6

$55.95 US
$83.95 CANADA